Efficient Matchmaking of Business Processes in Web Service Infrastructures

Vom Fachbereich Informatik
der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines
Doktor-Ingenieurs (Dr. -Ing.)
von

Bendick Mahleko

geboren in Chipinge, Simbabwe

Referent	: Prof. Dr. Erich J. Neuhold
Korreferent	: Prof. Kwei-Jay Lin, Ph.D.

Tag der Einreichung: 24. Mai 2006
Tag der Disputation: 11. Juli 2006

Darmstadt 2006
Hochschulkennziffer: D17

This thesis is dedicated to Sarah, Nyasha, Rutendo and Douglas.

Acknowledgements

Many people have contributed to the success of this work. I would like to thank my supervisor Professor Erich J. Neuhold who guided this research to this mature stage. The Intelligent Information Systems Institute (i-Info) group at Fraunhofer IPSI provided a conducive atmosphere to carry-out this research. I am grateful to the entire i-Info group for enabling this research and for the support they provided. I would like to thank Dr. Peter Fankhauser for helping to identify the topic and for supporting this work to the current stage. I am grateful to my mentor Dr. Andreas Wombacher for spending long hours with me, discussing and shaping this work to this stage. Last but not least, my thanks go to my wife Sarah for the endless support she provided.

Abstract

The problem addressed in this dissertation can be stated as follows: *given as a query, a business process description within a Web service infrastructure, efficiently find business processes from a large repository that complementarily support the input query.* In particular, the input business process enforces that some parts of the business process be mandatory. For example, a buyer organization using RosettaNet PIPs and the corresponding data dictionary might want to find suppliers that use the same RosettaNet standard that fulfill his business process. In this case, for instance, a mandatory cancellation procedure of the processing would be executed after the order had been placed with the supplier. The current Web service standards around Universal Description, Discovery and Integration (UDDI) and related technologies do not address this problem as they offer only service discovery based on attribute/ value queries, that is limited to atomic service discovery. Research has shown that Business Process Execution Language for Web Services (BPEL4WS or BPEL for short), which is currently used to express business processes in Web service environments, does not have a solid formal model and thus lacks formal semantics for querying business process descriptions. This means that a formal model is required to express business processes and to enable their querying. Based on this formal model, appropriate indexing techniques are needed for efficient querying in large service repositories.

A business process can be described as a set of message sequences each representing a potential execution sequence of the process. Based on this model, business processes are queried by finding matching business processes within the repository; the matching operation can be defined by computing a common set of message sequences covering the mandatory parts. The business process matchmaking semantics can be formalized by modelling business processes using an enrichment of finite state automata (FSAs) with propositional logical expressions. We call the enriched FSAs annotated finite state automata (aFSAs). aFSAs can express message sequences (potentially infinite) in a business process while logical expressions express mandatory as well as optional parts of the business process. A business process matchmaking is formally defined as the non-empty intersection of aFSAs, covering mandatory parts of the business process description.

Computing the intersection of aFSAs is computationally expensive being more than quadratic on the number of states and transitions. Further, sequentially scanning large service repositories to find matching business processes will not scale even if individual query operations were simple. The traditional way to speed-up query performance operations in databases is to use indexes. However, indexes for intersection-based queries on aFSAs are also not supported by standard database indexing techniques such as B+-trees. An option is to construct an index based on aFSA message sequences and annotations, and using message sequence equivalence

as well as the evaluation of annotations to realize intersection operations. However, the number of message sequences in an aFSA may be infinite due to cycles in the business process specification.

In this dissertation a formal model for indexing and querying business processes is developed. The indexing approach is based on abstractions to transform aFSAs via their grammars into forms that can be indexed by available indexing mechanisms like B+-trees. The indexing approach is implemented and evaluated. Evaluation results show that the index outperforms sequential scanning by several orders of magnitude. An analysis of evaluation results also shows that the index scales well with increasing data set sizes and exhibits good computational properties for searching, being approximately linear on the number of aFSAs in the data collection.

Deutsche Zusammenfassung

Die Problemstellung, die in dieser Dissertation adressiert wird, kann wie folgt beschrieben werden: *eine Beschreibung eines Geschäftsvorgangs, die in einer Web Service Infrastruktur in Form einer Abfrage vorgegeben ist, soll in einer großen Datenquelle effizient Geschäftsprozesse aufspüren, die komplementär zur Eingabe-Abfrage sind.* Insbesondere kann der Eingabe-Geschäftsprozess vorschreiben, dass einige Teile des Geschäftsvorgangs zwingend erforderlich sind. Zum Beispiel sucht ein Einkäufer, der eine bestimmte RosettaNet PIPs Beschreibung und das dazu gehörige Datenbeschreibungsverzeichnis einsetzt, einen Lieferanten, der ebenfalls demselben RosettaNet Standard folgt und somit auch den Geschäftsprozessen des Einkäufers entspricht. Nach Abschluss der Auftragsvergabe würde hier der Geschäftsvorgang zwingend beendet werden. Die derzeitigen, mit UDDI (Universal Description, Discovery and Integration) erstellten Webservice Technologien sowie verwandte Technologien sind nicht in der Lage dieses Problem zu lösen, da diese zur Dienstfindung nur begrenzte Unterstützung auf Attribute-/Wertebasierenden Abfragen anbieten können und demnach auf die atomare Dienstfindung beschränkt ist. In der Forschung werden BPEL4WS oder BPEL (Business Process Execution Language für Web Services) zur Beschreibung von Geschäftsprozessen verwendet. Es hat sich gezeigt, dass diesen Sprachen ein solides, formales Modell und daher auch eine formale Semantik fehlt, um Abfragen von Geschäftsvorgangsbeschreibungen durchzuführen. Dies bedeutet, dass ein formales Modell benötigt wird, um Geschäftsprozesse darstellen zu können und ihre Abfrage zu ermöglichen. Geeignete, auf einem solchen, formalen Modell aufbauende Techniken werden zur Indizierung benötigt, um Abfragen in großen Datenquellen effizient gestalten zu können.

Ein Geschäftsprozess kann als eine Abfolge von Nachrichten beschrieben werden, wobei jede Nachricht eine mögliche Sequenz in der Ausführung eines Vorgangs repräsentiert. Anhand dieses Modells kann durch Abgleichen der Geschäftsprozesse in der Datenquelle nach passenden Prozessen abgefragt werden. Der Abgleichvorgang ist definiert durch die Berechnung einer gemeinsamen Menge an Sequenznachrichten, die die zwingend notwendigen Teile enthalten. Die Semantik des Algorithmus zum Abgleich von Geschäftsprozessen kann durch Modellieren der endlichen Zustandsautomaten (final state automata, FSA), d. h. durch Anreichern mit aussagenlogischen Ausdrücken, formalisiert werden. Diese angereicherten FSAs (aFSAs) können Sequenznachrichten (möglicherweise unendlich) in einem Geschäftsprozess ausdrücken, während logische Ausdrücke sowohl erforderliche als auch wahlfreie Teile des Geschäftsprozesses darstellen können. Der Abgleich von Geschäftsprozessen ist formal definiert als die nicht leere Schnittmenge von aFSAs, die die erforderlichen Teile einer Geschäftsprozessbeschreibung enthalten.

Die Schnittmenge von aFSAs zu berechnen ist rechenintensiv und mehr als quadratisch zur

Anzahl der Zustände und übergänge. Ferner skaliert ein sequentielles Scannen von großen Datenquellen nicht bei der Suche nach passenden Geschäftsprozessen, auch wenn die einzelnen Abfrageoperationen einfach aufgebaut sind. Herkömmlicherweise werden Indizes verwendet, um Anfragen in Datenbanken zu beschleunigen. Indizes zu schnittmengen-basierten Anfragen von aFSAs werden jedoch von Standard-Datenbank-Indexierungstechniken wie beispielsweise B^+-tree nicht unterstützt. Eine Möglichkeit ist, einen Index auf Basis von aFSA Sequenznachrichten und Annotationen zu erzeugen, Hier werden Sequenznachrichten-äquivalenz und die Auswertung der Kommentierungen für die Erstellung der Schnittmengen mit berücksichtigt. Die Anzahl der Sequenznachrichten in einer aFSA kann unendlich sein, da sich Schleifen in der Spezifikation des Geschäftsprozesses bilden können.

In dieser Dissertation wird ein formales Modell zur Indexierung und Abfrage von Geschäftsprozessen entwickelt. Der Ansatz zur Indexierung basiert auf Abstraktionen zur Transformation von aFSAs über die Grammatik in eine Struktur, die durch verfügbare Indexierungsmechanismen wie z. B. B^+-tree indiziert werden kann . Die Indexierungslösung wird implementiert und ausgewertet. Die Evaluierungsergebnisse zeigen, dass die Indexierung das sequentielle Scannen um mehrere Größenordnungen übertrifft. Eine Analyse der Evaluierungsergebnisse zeigt darüber hinaus, dass der Index auch mit zunehmender Anzahl an Datensätzen gut skaliert und gute Recheneigenschaften bei der Suche aufweist, da die Suche sich linear zur Anzahl der aFSAs in der Datensammlung verhält.

Table of Contents

List of Figures

List of Tables

Nomenclature

$curP$	The current path
F	$F \subseteq Q$, the finite set of final states of a deterministic finite state automaton
G_n	A context sensitive grammar, with look-back of n
N_n	The set of non-terminals in a context sensitive grammar G_n
$L(A)$	The language of an aFSA A
Q	The finite set of deterministic finite state automaton states
q_0	$q_0 \in Q$, the start state of a deterministic finite state automaton
QA	$Q \times E$, is a finite relation of states and logical terms within the set E of propositional logic terms
$t(A)$	A function to transform an aDFA to that of a complementary role
$t_1(\Sigma)$	A function to transform messages Σ of an aDFA to those of its complementary role
$t_2(\Delta)$	A function to transform transitions Δ of an aDFA to those of its complementary role
$R()$	Number of production rules in an aFSA
$Reach()$	Reachability function
T_n	The set of terminals in a context sensitive grammar G_n
W_{sc}	Number of n-grams generated from a grammar of an aFSA with simple cycles
X^{q_i}	The set of outgoing transition labels of state q_i
Δ	The finite set of deterministic finite state automaton transitions
Φ_n	Transforms an aFSA language to a finite language, based on n-gram sets of the aFSA
P_n	The set of productions in a context sensitive grammar G_n
Π	Relational select operator

Ψ	A function to remove duplicate n-grams in an aFSA language abstraction
σ	Relational project operator
Σ	The finite set of deterministic finite state automaton messages
τ	A function to transform an aFSA language to a finite one, based on n-grams
aDFA	annotated Deterministic finite state automaton
BPEL4WS	Business Process Execution Language for Web Services
FSA	Finite state automaton
FSAs	Finite state automata
DNF	Disjunctive Normal Form
PIP	Partner Interface Process
RE	Regular Expression
UDDI	Universal Description, Discovery and Integration
UUID	Universal Unique IDentifier

1 Introduction

The rapid adoption of the Internet for doing business has resulted in more and more companies becoming global and relying on the Internet infrastructure and related technologies for doing business. This new way of doing business based on Internet-based systems has been coined as *Internet-based e-commerce* in some literature. A significant part of this new paradigm of doing business is finding suitable partners to do business with, in an automated and efficient way. To this extend, new standards and technologies have been developed to aid the automated (or semi-automated) discovery of business partners based on the services they provide. The Web service discovery technology is one of the latest technologies to be developed to support the discovery of service providers, using the services they provide. The Web service technology is based on the Universal Description, Discovery and Integration (UDDI) standard [Ariba et al., 2000], which offers a simple API description to search for simple services and their providers based on attribute/ value queries. In particular, attributes like business name, service name, key-ids, category-name, etc., are used as parameters by which discovery of matching service providers is performed. This approach to service discovery, is very limited in a number of respects. For example, more complex service descriptions of services such as process aspects, QoS aspects, semantics etc., are not supported. It is not possible using the current infrastructure to find service providers with matching business processes, simply because process semantics have not been deployed. In the last couple of years, more elaborate descriptions of services taking process information of services have been developed; as an example, Business Process Execution Language for Web Services (BPEL4WS) [Andrews et al., 2003]. Although this is a step in the right direction, the limitation with these process descriptions is that they lack solid formal semantics for service discovery [van der Aalst, 2003, Wombacher et al., 2004b].

In addition, service providers want to be able to express mandatory message sequences within their business processes. This means for example that the buyer may wish to find a seller who supports his business process in a complementary way, e.g., where cancellation and payment activities are obligatory, meaning the matching seller must support these messages. Such semantics are not easily expressible using the current descriptions and standards. Wombacher addressed the formal model description problem in his dissertation [Wombacher, 2005], but service discovery based on this model does not scale for large service repositories as it is based on sequential scanning of the entire repository to perform bilateral matchmaking above the database layer. Thus this model needs to be extended to handle large data sets, which are anticipated in real-life applications, in an efficient manner.

The ebXML infrastructure faces the same challenges. Like the Web service infrastructure,

the ebXML framework allows service providers to express their business capabilities (including their business processes), for example using trading partner profiles (CPPs). Collaboration protocol agreements (CPAs) between service providers and service consumers can be created on the basis of existing complementary CPPs from the repository. However, the ebXML framework provides no mechanism for matching CPPs to create CPAs [EBXML, 2005, Team, 2001b, Levine et al., 2001]. In the next section we further motivate the work presented in this dissertation.

1.1 Motivation

Organizations from various industries have invested in infrastructures and standards by which to carry out various business activities. This is true for sectors like banking (with standards like SWIFT [SWIFT, 2005]), the travel industry (with the OTA standard [OTA, 2003]) and the semi-conductor industry (with the RosettaNet [RosettaNet, 2005] standard). The RosettaNet standard defines the so-called Partner Interface Process^{TM}(PIPs) which are atomic processes that can be aggregated to form more complex processes. The PIPs are clustered into categories, among which are sub-processes such as ordering, shipping, after sales support and inventory management. Most business collaboration activities can be modeled using RosettaNet PIPs. In addition, RosettaNet defines a standard data dictionary such that all parties using the PIPs agree on the used semantics.

In this dissertation, existing standards like RosettaNet will be used to describe business processes since they are widely used in practice and there is industry-wide consensus about their semantics. Our assumption is that the parties intending to engage in a business collaboration are using a common standard like RosettaNet. This is a reasonable assumption because such standards exist and are widely used. Without an agreed standard one would have to engage in semantic reasoning, first to establish the concepts to which parties refer and to do semantic matchmaking thereafter. Semantic matchmaking of business processes in Web service environments is a specialized subject that we leave to experts from the semantic web community [Berbers-Lee et al., 2001, McIlraith et al., 2001].

1.1.1 Automating the Matchmaking of Business Processes

There are benefits to be derived by automating the discovery of potential business partners on the basis of their business processes. Even though standard data dictionaries for organizations in different vertical industries exist, most companies and organizations do not implement all parts of the standards. In most cases, these standards are huge, complex and modular such that organizations can focus on a subset of the specifications that they consider mission critical. Another reason for not implementing these standards fully is due to budgetary constraints. It is sometimes simply too expensive, for example, for a travel agent company to implement all message formats and business cases specified by the Open Travel Alliance (OTA) consortium.

If business processes have been matched, change in organizational structures and associated business processes is almost inevitable in the life of a company or organization. When these changes occur, it is necessary to recheck whether the business processes still match with the processes of their partners, including and in particular previously matching ones. Trying to determine which partners are supporting which parts of the standards without automated tools can be very complex, time consuming and error prone.

1.1.2 Web Services Infrastructure-based Service Discovery

The UDDI infrastructure [IBM et al., 2002] is the primary platform for service discovery in Web services environments. We present a buyer and seller example where the buyer wants to find sellers whose processes can successfully collaborate with his. The basic steps are that organizations register their profiles including information such as business identification, business category e.g., based on standard taxonomies such as DUNS business category number, geographical location (ISO 3166-1999), service descriptions and standards compliance. This information is

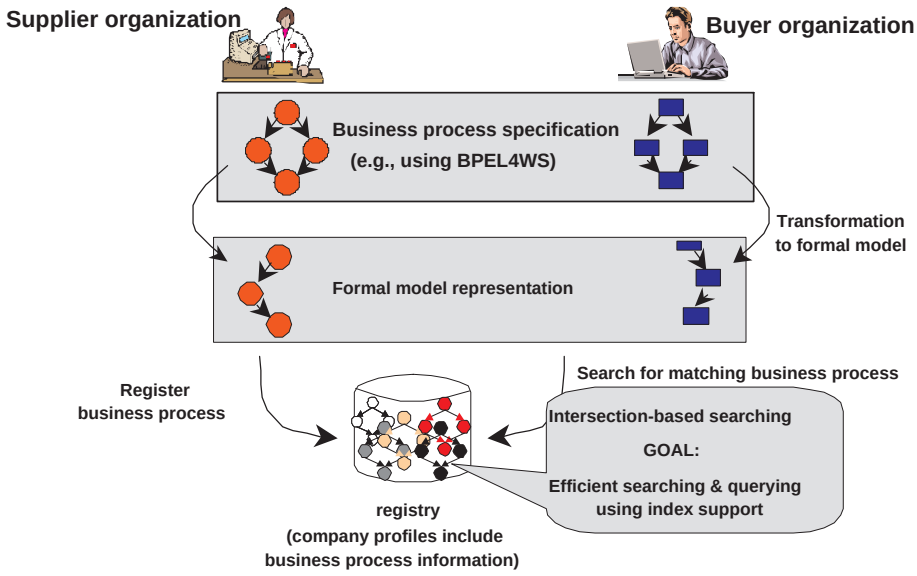


Figure 1.1: Buyer Looking for Sellers with Matching Business Processes

saved in a standard UDDI data structure. Organizations and companies looking for business partners search this repository using the standard UDDI search interface. The UDDI query is based on standard UDDI API, that uses a *find* service SOAP call [Gudgin et al., 2003]. As already pointed out earlier, UDDI queries are limited and can only support attribute/ value search, taking a string as input (for example name or category of business) and using string equivalence to compute the set of matching businesses or businesses belonging to the supplied category.

This is how service discovery is working today within the UDDI service discovery infrastructure. There is no support for more complex queries, like finding business partners on the basis of more complex search criteria such as business process descriptions.

The example in Figure 1.1 extends the basic UDDI profile by allowing companies and organizations to register and search for business partners on the basis of additional dimensions like business process descriptions. In this example, supplier or seller organizations register their business processes on a business registry. The business processes are transformed to a suitable formal model, that will later make it possible to do a search on behalf of the business processes. Buyer organizations or companies refer to any organization that references the business registry to search for business partners with compatible business processes. They provide their own business process which will be used to find matching business processes.

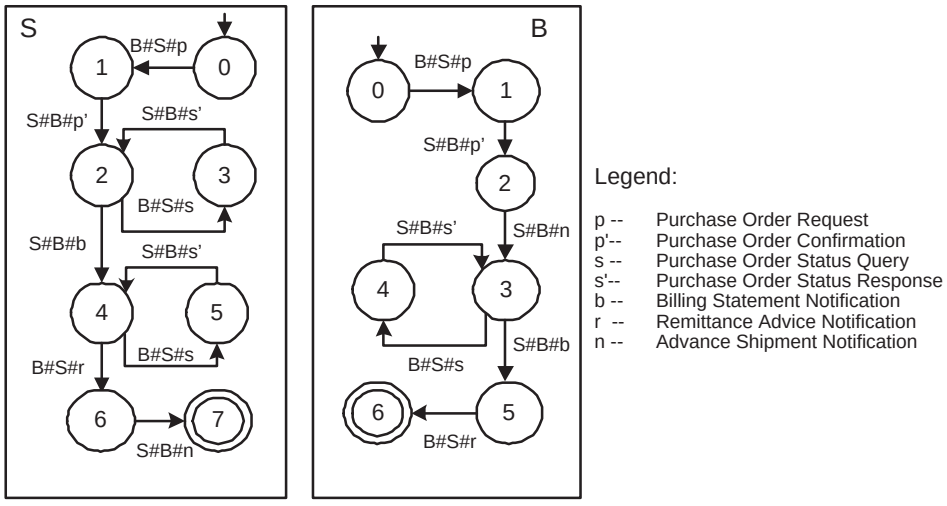


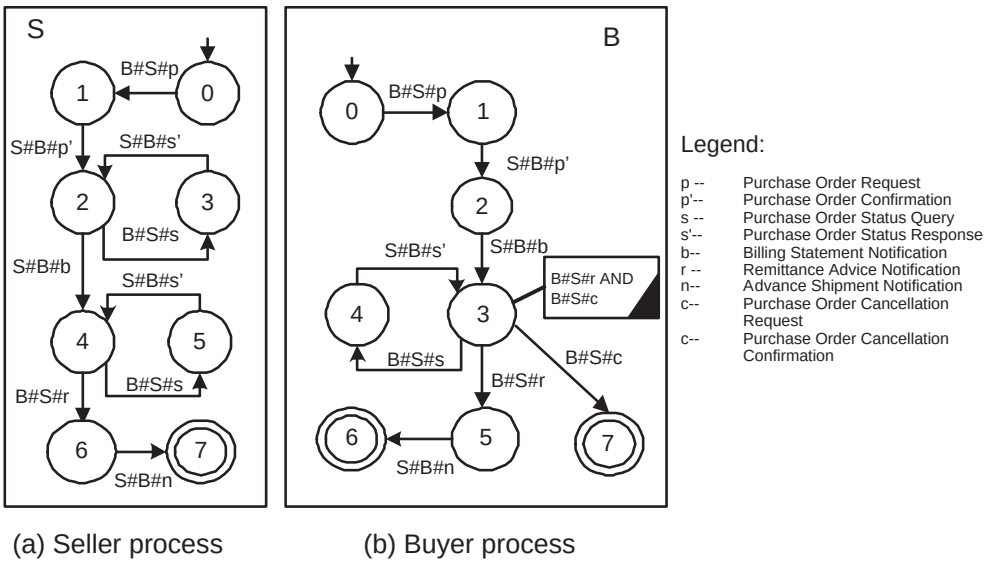
Figure 1.2: (a) Seller and Buyer Processes

The figure illustrates that the input query is not a simple attribute, but a business process. On the other hand the data is also a business process and matchmaking must proceed by comparing execution paths within these business processes from the start state.

Figure 1.2 is a simplified but illustrative example showing a seller *S* and a buyer *B*, represented by their business processes which are depicted as finite state automata. The finite state automata states represent business states. A finite state automaton state with an incoming transition with no source node is a business process start state. As an example, finite state automaton state 0 is a business process start state. Nodes with concentric circles represent final states, meaning, the sequence from the start state to this state represents a valid business process instance. Arcs connecting two states are transitions, representing a change in business state, triggered by a

business event such as receiving a purchase order. Arcs are labeled with messages denoted as *sender#receiver#messageName*, where *sender* represents the role that sends the message, *receiver* represents the role of the partner receiving the message in a bilateral collaboration and *messageName* is the message name. The roles and message names are all standardized based on existing standards such as the RosettaNet specification [RosettaNet, 2005].

The business process of a seller can be explained as follows: the seller expects to receive a purchase order request message *B#S#p* (PIP3A4) from a buyer and he responds with a purchase order confirmation *S#B#p′* message (PIP3A4). Having confirmed the purchase order, the seller can get a purchase order status query *B#S#s* (PIP3A5) to which he must respond with a purchase order status response message *S#B#s′* (PIP3A5). This query can be sent zero or more times. Having confirmed the purchase order, the seller expects to send a billing statement notification message *S#B#b* (PIP3C5) to the buyer. Again, in this state, the buyer may enquire about the status of his purchase order using a *B#S#s* message (PIP3A5). The seller expects that after receiving the billing notification *S#B#b* (PIP3C5), the buyer will send a remittance advice notification message, *B#S#r* (PIP3C6), showing his payment plan for the items ordered. Finally, the seller sends a notify of advance shipment message *S#B#n* (PIP3B2) informing the buyer about shipment.



(a) Seller process (b) Buyer process

Figure 1.3: Seller and Buyer Processes Including Mandatory Message Sequences

The seller and buyer business processes use exactly the same messages, but are ordered differently. The difference is that the buyer expects to get shipment information *S#B#n* (PIP3B2) soon after his purchase order is confirmed by the seller via *S#B#p′* message (PIP3A4) and do remittance via a *B#S#r* message (PIP3C6) later, while the seller process is sending shipping

information via an *S#B#n* message (PIP3B2) only after the remittance advice *B#S#r* message (PIP3C6) has been received. Thus from a message level the two processes match; however, when the order is taken into account, the two business processes do not match. Thus ignoring the message order and relying on a message set intersection results in a false match in this case.

In Figure 1.3, the buyer *B* insists on certain message sequences being supported as an additional condition for matchmaking. In this case, a matching seller *S* must support the remittance advice notification *B#S#r* and the purchase order cancellation *B#S#c* (PIP3A9) messages. Thus *B#S#r* and *B#S#c* are mandatory for a trading partner. This is depicted by a logical expression *B#S#r* AND *B#S#c* on state 3 (Figure 1.3 (b)). The standard semantics of automata is an optional execution of transitions. As an example, in Figure 1.3 (a) the two transitions $2 \xrightarrow{s} 3$ and $2 \xrightarrow{b} 4$ are optionally traversed to leave state 2. Thus using the representation shown in Figure 1.3, the buyer can express mandatory message sequences to be considered for matchmaking. In Figure 1.3, the buyer and the seller, though sharing at least one message sequence in common, do not match because of the mandatory cancellation message of the buyer, which is not supported by the seller. In order to match these processes, we need to evaluate the logical expressions as well as compare the message sequences for a match.

In both Figures 1.2 and 1.3, the seller and the buyer processes are cyclic. This implies that the number of message sequences from each of the partners is infinite. It follows that existing database indexing techniques cannot be used for matching business processes.

This dissertation presents a methodology for indexing and matching business processes like those depicted in Figure 1.3 in order to facilitate efficient service discovery. The problem introduced in the present section is not limited to Web services, but can also be observed in other environments of electronic data interchange. The ebXML is one such environment.

1.1.3 ebXML Framework - Business Process Matchmaking

The ebXML standard [Chappell et al., 2001, Kotok and Webber, 2001, EBXML, 2005] is an initiative of the United Nations Center for Trade Facilitation and Electronic Business (UN/CEFACT), with the Organization for the Advancement of Structured Information Standards (OASIS) as technical partner. The goal of ebXML is to create a single global marketplace where businesses can find each other, agree to become trading partners, and conduct business [Hofreiter et al., 2002]. To this end, the ebXML standardization committee has developed several standards.

Among the standards, is the Business Process Specification Schema (BPSS) standard [Team, 2001a] which allows trading partners to describe their business processes. The description of business processes by trading partners using a single standard enables the execution of business collaborations among the partners. The BPSS standard supports the basic constructs for creating processes such as sequence, branching, parallelism, join and repetition. Based on the control flow constructs, BPSS allows the description of binary collaborations. Also multi-lateral collaborations can be synthesised from binary collaborations and represented from a global point of view. Thus with BPSS, potential trading partners can describe bilateral or multilateral collaborations and store them in a repository, for future reference and as a basis for establishing a

business collaboration agreement.

The challenge is how to automate the process of creating collaborations from existing descriptions. An approach related to the ebXML initiative is the Collaboration-Protocol Profile and Agreement Specification [Aissi et al., 2002]. This specification allows each trading partner to describe his or her trading profile in a Trading Partner Profile (TPP). The trading profile of a trading partner describes that partner's capabilities, including the supported business processes which are described using BPSS. The agreed intersections between the profiles of two business partners are documented in a Trading Partner Agreement (TPA) [Hofreiter et al., 2002]. The message exchange capabilities of a trading partner are captured in a Collaboration Protocol Profile (CPP) within a TPP. Also, the agreed message exchange capabilities between two partners are captured in a Collaboration Protocol Agreement (CPA) within a TPA.

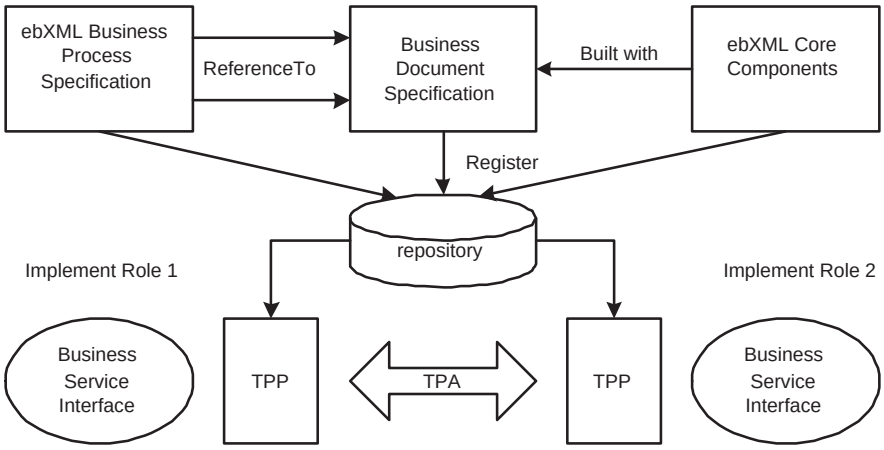


Figure 1.4: ebXML BPSS and ebXML CPP/ CPA Specifications

Figure 1.4 [Team, 2001a] shows the relationship between the ebXML standards and initiatives related to ebXML standards as described in this section. Business processes are built based on the BPSS standard, making use of ebXML core components, and stored in a repository. Individual business partners can build their own TPPs, by referencing process specifications stored in the repository. A TPA is built by computing the intersection of the trading partners' TPPs that have complementary roles.

The problem, how to automate the creation of such TPAs from TPPs is not yet addressed. We can treat this as a search problem, whereby one of the trading partners submits his TPP and would like to find trading partners to collaborate with; i.e., the business partner would like to find business partners whose TPPs match with his. Matching partners are found by computing the intersections of the TPPs in the repository with those of the partner performing the search. Efficient computations of such intersection queries are not supported by existing databases and indexing techniques. The dissertation addresses these issues by describing formal semantics for

matching business processes and an efficient implementation of these semantics.

1.2 Problem Statement

The problem to be addressed in this dissertation is how to do efficient service discovery using more complex criteria, than provided by the UDDI standard. These criteria lead to *efficient matching of business process descriptions* of services. The main challenge is how to come up with a formal indexing method to efficiently perform discovery of business process enriched services in large service repositories. The discovery of business process enriched services requires new semantics for matchmaking because the simple attribute/ value search approach (UDDI) does not apply to business process descriptions since they comprise complex structures which must be compared in order to find a match. As we pointed out earlier, the semantics for matching services enriched with business process descriptions do not rely on the traditional simple attribute/ value approach, where an index can easily be defined using one of the attributes as the index key. Moreover, the matching criteria is not simply an equivalence of service attributes, but we need to consider the structural information of the business processes involved. Thus more complex indexing mechanisms are required for efficient discovery of services that use business process description information for discovery.

1.3 Matching Business Process Descriptions

Matching business process descriptions is based on the atomic pieces of the processes. These atomic processes comprise a description of incoming and outgoing messages as well as which roles are involved. Thus for a given collaboration scenario, messages that are sent by one role must be received by another role (playing a partner role) such that a business objective such as procurement is fulfilled for a match to occur.

We assume in this dissertation that the involved business partners are using a common data dictionary, that is the same task descriptions have the same semantics. This is a reasonable assumption because standard bodies already exist to define message formats and their semantics, like RosettaNet [RosettaNet, 2005], Open Travel Alliance [OTA, 2003] and IOTP [Burdett, 2000]. Thus, no semantic reasoning is required when matching business processes, as they derive from the same data dictionary.

Business processes are composed from simple services to create value chains. These services typically need to be processed in a particular order. The processing order can be expressed using standard process description languages like Business Process Execution Language for Web Services (BPEL4WS) [Andrews et al., 2003]. Each simple service represents a particular activity of either sending or receiving a message. Thus, by carefully selecting the involved simple services and their processing order, explicitly representing the start and terminating services, meaningful business processes are constructed, representing activities of a business partner such as order fulfillment. In this dissertation, the set of message sequences, with well-defined starting and

terminating messages that are sent and/ or received by a business partner via simple services to fulfill specific business goals, describe a business process.

Some business process specifications can be cyclic. Consider an example where, after sending a purchase order, the buyer organization might want to continually check for the status of the order. Thus, after checking for the status, the buyer process remains in the same business state, e.g., if a particular response is received. Such a situation is expressed by a loop, which results in a cyclic business process specification. Cyclic business processes can also occur if, when traversing a path defined by a message sequence will result in some parts being revisited.

In summary, two business processes belonging to two complementary roles like a service provider and a service consumer only match if they have at least one message sequence in common, including all mandatory messages. This means that the two parties are able to successfully conduct a business collaboration to fulfill a specific business objective such as order fulfillment. If two partners share a common message sequence, every message (from the common sequence) that is sent by one partner is successfully received by the complementary partner and vice-versa, and all mandatory messages are covered.

1.4 Indexing Business Processes

The business process matchmaking semantics must be implemented efficiently to support service discovery in large service repositories like on the Internet scale. Indexing support is needed in large service repositories since the number of business processes to be compared is large, thus scanning whole repositories to compute intersection (more than quadratic computational complexity) is computationally expensive. Building an index requires a mechanism to represent message sequences (potentially infinite) that describe business processes. The queries are also message sequences (potentially infinite) describing business processes to be matched.

1.5 Research Challenges

Research challenges in this dissertation fall into three clusters:

- formal model definition for the index,
- indexing infinite languages with mandatory message sequences.

Since existing business process descriptions lack solid formal models, the semantics for matching business processes is not defined [van der Aalst, 2003]. Wombacher presented formal semantics for bilateral matching of business processes in his thesis [Wombacher, 2005]. This model can be used for the sequential matching of business processes, but for use on large data sets as anticipated on Internet service repositories, the model must be extended to support indexing for efficient search operations. The challenge is to find a suitable formal model to represent the index and expressing formal query definitions and associated operations. The indexing model

must express mandatory execution paths within a business process, in addition to the representation of (potentially infinite) sequences. Representing mandatory execution paths of a business process specifies which execution paths of the business process must be fulfilled by the matching partner, thus enforcing the matching partner to support those parts of the business process deemed critical for the service finder.

Business processes can be represented as sets of message sequences representing a (potentially infinite) language. While it is possible to store business processes for indexing finite languages, it is not possible to store infinite languages. The infinite language is a result of cycles in business process specifications. The challenge is how to represent business processes finitely in the database for indexing infinite languages. Another challenge is how to represent mandatory message sequences without losing information. Not only data is infinite, but the query represents potentially an infinite language, with mandatory execution paths. The challenge is how to query using this infinite language with mandatory parts.

Queries are evaluated by computing the intersection of the input language and those from the repository, to cover all mandatory parts, followed by checking for non-emptiness of intersection results. Intersection[1] as a query operator is not supported by databases. The challenge is how to transform the intersection operator to operators supported by databases (for example string equivalence), without losing the intersection semantics.

1.6 Contributions and Main Results

This dissertation makes two major contributions (i) development of an indexing technique (and its formalization) for efficiently querying business processes within a Web service environment and (iii) implementation of a Web Service Matchmaking Engine based on the indexing approach using annotated finite state automata to represent business processes.

1.6.1 Indexing Techniques for Annotated Finite State Automata

In this dissertation a technique for indexing annotated finite state automata (aFSAs) to support intersection queries is developed [Mahleko et al., 2005b, Mahleko et al., 2005a]. aFSAs, which are extensions of finite state automata, are used in the dissertation to model business processes with mandatory message sequences. The new indexing technique relies on existing structures such as B^+-trees for data storage and access and can thus be deployed on existing Web service infrastructures. The new indexing technique is evaluated both analytically and experimentally and results show a significant performance gain over sequential scanning for aFSAs with simple cycles.

[1] Language intersection, not to be mixed-up with set intersection

1.6.2 Implementation of a Web Service Matchmaking Engine based on aFSA Index

The indexing technique for aFSAs, developed in this dissertation has been implemented as part of the IPSI-PF business process matchmaking engine [Wombacher et al., 2004c]. This engine allows to find matching business partners based on their business process descriptions and such attributes as category information which relies on UDDI. Thus the engine extends UDDI with support for process matching semantics based on a scalable indexing mechanism.

1.7 Thesis Outline

The rest of this dissertation is structured as follows: Chapter 2 presents an analysis of related work. Related work is clustered into three parts: (i) related work pertaining to existing service discovery techniques, (ii) related work pertaining to formal models for matchmaking business processes, and (iii) related work pertaining to indexing techniques for business process matchmaking. This describes the different clusters of related work and ends with an assessment of the work.

Chapter 3 presents a formal model for describing business processes. In this chapter, the definition for matching business processes is given. This definition is based on an enrichment of ordinary finite state automata with logical expressions to annotate outgoing transitions of a state as optional or mandatory. This type of FSA is termed annotated finite state automata (aFSA).

Chapter 4 presents an indexing technique for aFSA to answer intersection queries. The indexing technique has the following goals: (i) exclude false misses i.e., finds all existing matches (ii) minimize false matches i.e., allow false matches, but maximize the quality of search results (iii) increase search performance i.e., reduce the search space to a minimum. In addition, the index relies on already deployed indexing structures like B^+-trees.

Chapter 5 focuses on implementation and evaluation. The chapter presents a realization of the matchmaking approach and index within a Web service application environment. It illustrates how the current UDDI infrastructure can be enriched with complex business process matchmaking semantics, rather than the existing simple attribute/ value match. The chapter also presents experimental and analytical evaluation results for the matchmaking approach.

Chapter 6 summarizes the results of this dissertation, outlines the main contributions of the dissertation and provides an outlook on future work.

2 Analysis of Related Work

Work related to this dissertation can be clustered into three main categories: (i) related work with respect to existing service discovery techniques, (ii) related work with respect to formal methods of matchmaking and (iii) related work with respect to existing indexing techniques.

2.1 Service Discovery Techniques

This section describes the UDDI standard for describing services within the Web service infrastructure and some extensions. Existing tools based on the approaches are also summarized. An assessment of existing techniques is given at the end of the section.

2.1.1 UDDI Specification

The UDDI specification has become the de facto standard for service discovery in Web service infrastructures [IBM et al., 2002]. It comprises three main parts: (i) "white pages" - which are used to hold basic information about a company or organization e.g., address, contact information, company description, (ii) "yellow pages" - which allow companies and organizations to be listed in UDDI registries based on industry categories that use standard taxonomies like UNSPSC (Universal Standard Products and Services Classification [UNSPSC, 2005]) or NAICS (North American Industry Classification System [NAICS, 2005]) and (iii) "green pages" - which allow companies and organizations to record interface details of how a service is to be invoked [ShaiklAli et al., 2003, IBM et al., 2002].

Service discovery on current UDDI implementations is realized by searching the UDDI registry using a limited number of service attributes as parameters. A simple service can be searched by service name, key reference or category bag. The service name is provided by the service provider during the registration of the service, and using *UDDI find_xxx* methods, a service matching this name can be found using string equivalence. A key reference is a unique identity that is given to a service, during service registration. By providing the key reference of a service (if known in advance, which is very unlikely) to a *UDDI find_xxx* method, the respective service can be discovered. A category bag comprises all the business categories in which a service has been listed. This means that a service can also be found by searching based on categories, that are described using standard taxonomies, through the *UDDI find_xxx* methods of the UDDI Inquiry API.

The three UDDI attributes used for service discovery in UDDI registries have severe limitations. While they allow service discovery of simple services which are stateless, they are not

capable of matching complex business processes, i.e., stateful Web services that must be executed in a certain order. The reason for this is that UDDI has no semantics for expressing business processes and matching them.

2.1.2 UDDI Extensions

There have been various proposals to extend the UDDI standard to address its limitations. Two main approaches have been used (i) enrich UDDI service descriptions with additional attributes that are used to enhance service discovery at the syntactic level only (ii) enrich UDDI service descriptions with semantic information that is used for service discovery.

Examples of UDDI extensions that enrich Web service descriptions with additional attributes at the syntax level only are UDDIe [ShaiklAli et al., 2003],

The semantic Web community has been very active in defining standards for extending Web service descriptions. The OWL-S coalition [Coalition, 2004] proposed an ontology for describing Web services based on the Web Ontology Language (OWL) [Dean and Schreiber, 2004]. OWL-S is structured into three types of knowledge: service profiles, service model and service grounding. Service profiles describe the capability of a Web service. The service model describes services in terms of inputs, outputs, preconditions and effects of invoking a service; processes in OWL-S are described in terms of their states, including information such as initial activation, execution and completion. Service grounding describes how to access the service.

OWL is derived from the DAML+OIL Web Ontology Language
[Horrocks et al., 2001, Dean and Schreiber, 2004]. DAML+OIL is based on Description Logic [Gonzalez-Castillo et al., 2001], thus descriptions written in DAML+OIL can be compared semantically [Li and Horrocks, 2003]. By describing service capabilities using OWL-S (which is based on OWL), and using a Description logic reasoner such as Racer [Haarslev et al., 2005] to semantically match Web service capabilities, it is possible to find matching Web services from a semantic perspective. Several Web service matchmaking prototypes have been implemented using this approach, for example [Chiat et al., 2004, Li and Horrocks, 2003, Paolucci et al., 2002, Trastour et al., 2001].
Work related to this is also found in [Patil et al., 2004, Sivashanmugam et al., 2003], where approaches for annotating Web services with semantic information, and using this for service discovery are described.

Technology around the Web Service Modeling Ontology (WSMO) [Roman et al., 2005], Web Service Modeling Language (WSML) [Bruijn, 2005] and Web Service Modeling eXecution environment (WSMX) [WSMX, 2005] allows Web services to be described semantically to facilitate the automated discovery, composition and invocation of services. Although WSMO allows Web services to be orchestrated and choreographed into more complex Web services, the specification does not provide a mechanism for automated matchmaking of complex services to establish bilateral or multi-lateral collaborations. Assuming that two WSMO descriptions belonging to different organizations are given; it is not possible to automatically check if the involved organizations can collaborate by comparing the message sequences they exchange.

There are also earlier projects such as infosleuth [Nodine et al., 2000, Nodine et al., 1999] and RETSINA/ LARK [Sycara et al., 1999b, Sycara et al., 1999a] that do not rely on OWL-S, but use the same basic principles of describing service capabilities using ontologies, and performing matchmaking by comparing capabilities. There have been other proposals to replace the current UDDI specification. The Web service specification (WS−Specification) is one such proposal [Overhage and T., 2002]. Like UDDI, it has the following parts to describe Web services: (i) "white pages" which describe non-functional aspects of a service like service name, description, global key, etc., performance and security information, (ii) "yellow pages" describe the application domain of a service and (iii) "green pages" describe technical information of service interfaces. In addition, "blue pages" are introduced, which describe conceptual semantics and pragmatics of a Web service. They allow semantic specifications of a service to be described, a feature that is not supported in UDDI. Although the WS−Specification is a very detailed document specifying how to overcome limitations of the UDDI specification, it is not currently implemented. WS−Specification also does not specify how to find service providers on the basis of their business processes.

The problem with the above-mentioned semantic-based approaches is that, while they are very efficient at finding simple services by matching semantic descriptions of their capabilities, it is not clear how to translate this to match complex business processes to check for bilateral or multi-lateral collaborations. OWL-S relies on service profiles to compare Web service capabilities, with process ontologies of service models not being used to match business processes. The same principles also apply to WSMO and related standards - given two WSMO descriptions belonging to different organizations, it is not possible to automatically check if the involved organizations can collaborate by comparing their message sequences.

2.1.3 Web service search engines

There has been a proliferation of Web service search engines on the Web in recent times. These can be clustered into two types. The first type accepts as input, keywords, which they use to search within WSDL descriptions of services. Examples of such Web search engines are Binding point [Homepage, 2005a], Grand central [Homepage, 2005b], Sal central [Homepage, 2005c] and Web service list [Homepage, 2005e]. The second type of Web service search engines goes beyond naive keyword matching of WSDL contents by performing a similarity search on WSDL operations of Web services, considering operation name and input/output parameters. Clustering techniques are used to match parameters from different Web services, thus ensuring that parameters referring to the same concepts are matched. An example that uses such a technique to find matching Web services is Woogle [Homepage, 2005d, Dong et al., 2004]. The approaches used by Web service search engines can only match simple services, thus do not handle the process aspects of services.

Table 2.1: Service Discovery Techniques Assessment

Technique\Web service	Stateless	Stateful	Semantic
UDDI	✓	×	×
UDDIe	✓	×	×
OWL-S	✓	×	✓
WSMO	✓	×	✓
Binding point	✓	×	×
Grand central	✓	×	×
Sal central	✓	×	×
Web service list	✓	×	×
Woogle	✓	×	✓

2.1.4 Assessment of Web Service Techniques

Table 2.1 summarizes service discovery techniques and the types of services discovered. The checkmark symbol (✓) means that the property on that column is supported by the standard. The cross symbol (×) means that the property is not supported. As an example, in the table, UDDI supports stateless services but does not support stateful services as well as semantic descriptions. In the table, stateless services are basic services that send a request and receive a response. They do not maintain state between service requests. Stateful services are services that represent several services that are executed in order. These services are described as business processes which expresses the order in which services are executed. Table 2.1 shows that none of the existing services is able to perform service discovery on the basis of business process descriptions.

2.2 Formal Models of Matchmaking

Formal models for matching business processes must support the intersection operation as well as emptiness testing of the intersection results. In addition, they must be capable of expressing not only optional messages, but also mandatory ones. Candidate formal models are process algebra-based models, Petri Net models, graph models using simulation/ bisimulation, graph models using sub-graph isomorphism, finite state automata and IOAutomata.

2.2.1 Process Algebra Models

Processes can be modeled using process algebras. Examples of process algebra models are calculus of communicating systems (CCS) [Milner, 1982], communicating sequential processes (CSP) [Hoare, 1985] and π-calculus [Milner, 1999]. These languages provide algebras for specifying and reasoning about concurrent systems or processes. They provide sets of terms, operators and axioms for writing and manipulating algebraic expressions. The behavior of the systems

being modeled can be analyzed based on the defined operators. The π-calculus, in addition to modeling concurrent systems, can also express mobile processes and techniques for analyzing their behavior. Some of the operations supported by these languages are simulation and bisimulation of process instances. For example, given two process descriptions in π-calculus notation, it is possible to check for their equivalence using bisimulation operation. In our problem statement we have stated that we are interested in checking for intersection and doing emptiness testing. None of these operations is directly supported by these process algebra-based models. The models also do not provide constructive mechanisms for carrying out intersection operations as provided for by finite state automata.

2.2.2 Petri Net Models

A Petri net is a formal and graphical language for modeling systems or processes [Peterson, 1981]. It comprises places, transitions, arcs and tokens. Places represent states of a Petri Net and they can contain tokens. Input arcs connect places with transitions, while output arcs start at a transition and end at a place. The current state of the modeled system, called the marking, is given by the number of tokens in each place. The system marking changes when transitions fire, meaning tokens are removed from input places and inserted to output places of a transition. Transitions can only fire if they are enabled, meaning, there are tokens ready to fire in the input places. Petri Net models support several operations, and in particular, they are closed under intersection. However the emptiness test and reachability problems of Petri Nets has been shown to be NP-complete [Esparza, 1998b, Esparza, 1998a]. Thus they are not suitable for modeling our problem. A related formalism to Petri Nets is the Workflow Net (WF-Net) [van der Aalst and Hee, 2002]. WF-Nets have been used to model interorganizational workflows in [van der Aalst, 1999]. Like Petri Nets, WF-Nets are token-based and contain places and transitions, but in addition, they contain a single initial and final place. They have better computational properties than Petri Nets, but are used in asynchronous communication models were messages may arrive in a different order to that in which they were sent. However in our model, we assume a synchronous communication model were messages arrive in the order in which they were originally sent.

2.2.3 Graphs Bisimulation/ Simulation Models

Another modeling approach is to represent business processes as graphs and use graph simulation/ bisimulation for matchmaking. The nodes of the graph will represent business states, while directed graph edges connecting nodes, are labeled with messages. Thus a transition from one graph node to another represents either sending or receiving a message. The graph can be marked with a designated start node and a set of final or accepting nodes. Two graphs can be compared or matched if their start nodes have a bisimulation relationship, meaning the two graphs exhibit the same behavior or are equivalent at each node [Henzinger et al., 1995]. Alternatively, the less restrictive simulation operation might be used for matching the graphs. Although this approach

might appear to be interesting at first, it has the limitation that the simulation operation does not have the properties of intersection, such as symmetry. On the other hand, if the more restrictive bisimulation operation is used, this amounts to saying the graphs being compared must be equivalent, whereas with intersection, a single common path is sufficient to define a match. The model also doesn't model mandatory semantics for message sequences.

2.2.4 Graph Matching

Another candidate approach is similar to that described in the previous section, except that the matching operation is graph matching [Cordella et al., 1998]. To find matching business processes, a query graph is matched with the stored graph using one of several graph matching algorithms. However the problem with this modeling approach is that first the sub-graph isomorphism problem is well-known to be NP-complete [Cordella et al., 1998, Bunke, 2000], and second, this matching semantics is different from intersection in that with intersection, message sequences along an execution path are considered while with graph matching, graph nodes are compared.

2.2.5 Finite State Automata (FSA)

An FSA is defined by a finite set of messages, states, a set of transitions, a start state and a set of final or accepting states [Hopcroft et al., 2001]. It can be represented as a graph with a single start state, where nodes represent states, arcs represent transitions connecting two states. Transitions are labeled with messages drawn from the message set. FSA graphs are traversed from the start state. Final states are specifically marked with concentric cycles and they represent the acceptance of a message sequence by the FSA. FSAs are closed under intersection and have polynomial time algorithms for emptiness test and intersection. However FSAs in their original form cannot represent mandatory semantics of message sequences.

2.2.6 Assessment of Formal Models

Table 2.2: Formal Models Assessment

Technique\ Property	Intersect	Emptiness test	Polynomial intersection	Polynomial Emptiness	Optional / mandatory semantics
Process Algebra	×	×	N/A	N/A	×
Petri Net	✓	✓	✓	×	×
Graph (Bi)simulation	×	×	N/A	N/A	×
Graph Matching	×	×	N/A	N/A	×
FSA	✓	✓	✓	✓	×

Table 2.2 summarizes formal modeling techniques presented in this sub-section. The columns represent the properties of intersection *Intersect*, emptiness testing *Emptiness test*, whether the complexity of performing intersection is polynomial *Polynomial intersection*, whether the complexity of performing emptiness testing is polynomial *Polynomial Emptiness* and the ability to express mandatory message sequences. As already described, the checkmark symbol (✓) means that the property on that column is supported by the technique while the cross (✕) means that the property is not supported. In addition, the symbol N/A means not applicable, meaning the property does not apply to the technique with which it has been associated. From this table, it is clear that none of the formal models can express mandatory message sequence semantics, hence the need for a new model for matching business processes. Since our goal is indexing we need a model having at least a feasible complexity. From the analysis above, the model with the most feasible complexity is the finite state automata. Thus we will extend finite state automata to handle mandatory message sequences.

2.3 Indexing Techniques

Related work pertaining to indexing techniques can be broadly clustered into: (i) traditional indexing techniques, (ii) object oriented databases (OOD) indexing techniques (iii) graph-based approaches (iv) set indexing (v) semi-structured data and (vi) RE-trees.

2.3.1 Traditional indexing techniques

The most widely used external memory indexing structure in commercial relational databases is the B^+-tree. B^+-trees are hierarchical search structures
[Gray and Reuter, 1993, Comer, 1979, Ullman, 1988, Knuth, 1998]. They are based on multi-way trees [Sedgewick, 1998] with two types of nodes: index nodes (also called internal nodes) and leaf nodes. Index nodes store routing information that is used for navigating the tree during search operations while leaf nodes store tuples of relations (data records) or pointers to tuples of relations. Each index node contains a sorted sequence of key values that divide the search space covered by the node.

Searching on a B^+-tree is keyword-based comparison and is accomplished by navigating a path from the root to the leaf node. The search is directed by index nodes in which the search key value is compared to key values in internal nodes to determine the tree branch to follow. If the search key value is stored, it is found in one of the leaf nodes; otherwise the search may stop in one of the index nodes. Nodes are implemented as pages that can be loaded into main memory [Gray and Reuter, 1993, Vitter, 2001]. So when an appropriate page has been found, it is necessary to scan the page to find the record that is the subject of the search.

The B^+-tree cannot be used directly to index business process models, with infinite message sequences, because it does not support intersection operations as well as the emptiness test.

2.3.2 OOD indexing techniques

Data in object oriented databases (OOD) is organized as graphs where vertices represent objects and edges connect objects that reference one another [Bertino and Foscoli, 1995, Maier and Stein, 1986, Gyssens et al., 1990]. This section examines common indexing techniques for object-oriented databases and how they can be related to indexing business process models.

Indexing techniques for object oriented databases can be grouped into *aggregation* and *inheritance* indexing techniques
[Bertino et al., 1998, Bertino, 1994, Banerjee et al., 1987, Kang et al., 2000]. Aggregation indexing techniques are about indexing objects based on object attributes that reference other objects; inheritance indexing is about efficient evaluation of queries based on inheritance hierarchies.

Aggregation Indexing

The principle behind aggregation indexing is to materialize object reference chains from the root or non-root object to the object to be retrieved.

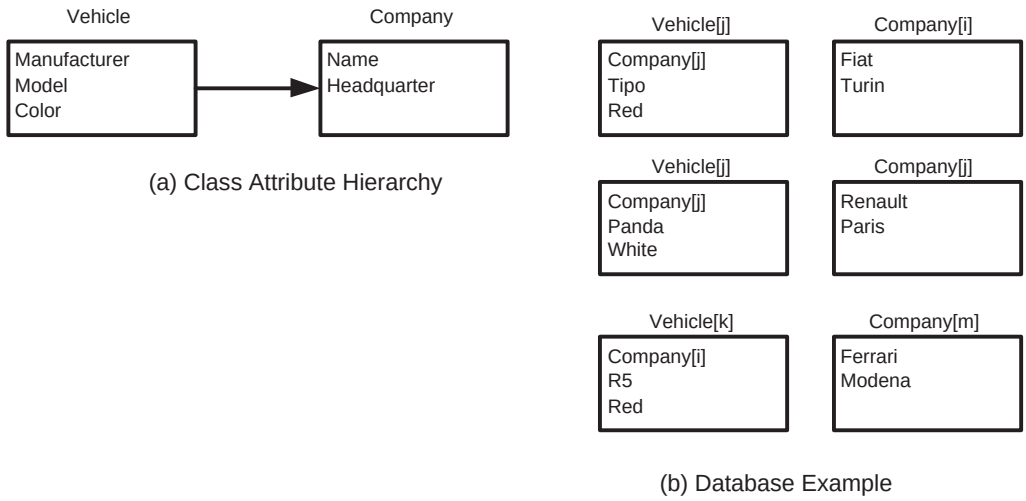


Figure 2.1: Nested Index Example

The nested index (NX) technique keeps a record of all objects within the aggregation hierarchy, that reference (either directly or indirectly) the object acting as index key
[Bertino and Kim, 1989, Chawathe et al., 1994, Gudes, 1997, Jiang et al., 1994]. The index can be abstracted as a pair (O,S) where O is the index key and S is a list of object *ids* that directly or indirectly reference the index key O. Figure 2.1 is an example showing a simple class attribute hierarchy and an example database. The example is based on the work of [Bertino and Kim, 1989].

In this example, nested index associating the *Name* attribute with object identifiers of *Vehicle* where the manufacturer is an instance of the company whose name is the key value is given as $\left\{(Fiat, \{Vehicle[k]\}), (Renault, \{Vehicle[i], Vehicle[j]\})\right\}$. In this case O which represents the *Name* key value has value *Fiat* when Vehicle is *Vehicle[k]* meaning *Vehicle[k]* object is referenced by key value *Name*. The same applies to the Renault key value.

The nested index is implemented using a B^+-tree: index nodes (internal nodes) contain among other information, the key value O and a pointer to route the search; leaf nodes contain among other information, the key value (the value of O), the number of object ids and the list of object ids that reference the key O.

This indexing technique does not support intersection operations. There is also no mechanism to handle infinite object hierarchies, which would correspond to infinite message sequences in our problem domain.

Inheritance Hierarchy Techniques

The access scope of a query against a class in object oriented databases is either instances of that class alone or instances of that class and its subclasses due to the effect of class inheritance relationships. Such relationships can be modeled as graphs where nodes represent classes and their attributes and edges represent inheritance relationships between classes. Techniques in the OOD community have been developed to index such data and relationships.

The class hierarchy index (CH-index) is used where there is an inheritance hierarchy relationship among classes
[Kim et al., 1989, Gudes, 1997, Ooi et al., 1996, Ramaswamy and Kanellakis, 1995]. It is based on the premise that rather than having an index for each class in an inheritance hierarchy, it is more efficient to have a single index for the whole class hierarchy; the result is all entries with the same keys are grouped together, irrespective of their originating class in the inheritance hierarchy.

The CH-index is implemented using B^+-trees. The B^+-tree implementation of a CH-index can be characterized as follows: index nodes contain among other information, index key values and pointers to next-level nodes; leaf nodes contain index key values as well as directories. Directories are used to group together objects with the same key values according to their classes. The index can be viewed as $\{(key\,value, \{class\,id, [o_1, o_2, \cdots, o_n]\})\}$ where $key\,value$ is the key index value within the class hierarchy, $class\,id$ is the id of a class in the class hierarchy and o_i is an object instantiated from class with id $class\,id$, which also belongs to the same class hierarchy.

This kind of index has no mechanism to handle infinite object structures which correspond to infinite message sequences and does not support intersection operations.

2.3.3 Graph-based approaches

Graph based approaches work by building a fingerprint of the database by fixing the maximum path length - also called limited path length. The GraphGrep graph indexing technique

uses such an approach. It is used to find all occurrences of a graph in a database of graphs [Giugno and Shasha, 2002, Shasha et al., 2002]. It is based on a hash-based fingerprinting technique to represent graphs. The graph is indexed using a hash table to persistently store paths in the graph, where the keys to the hash table are hash values of label paths.

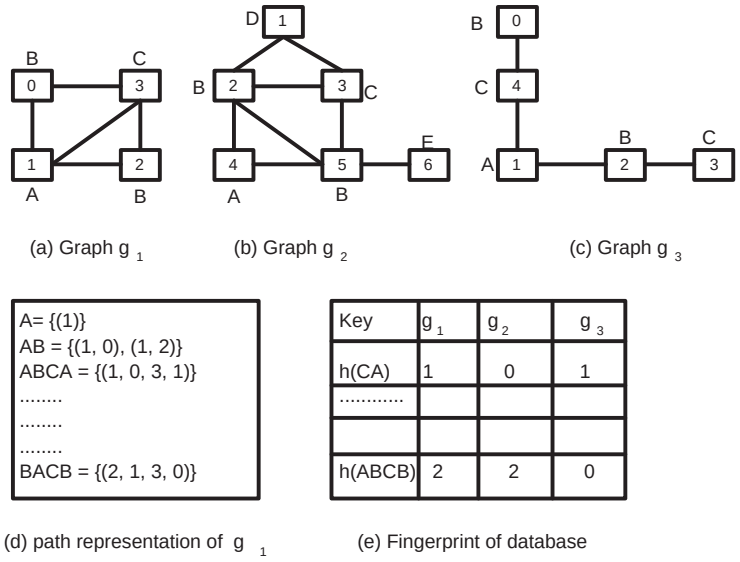


Figure 2.2: GraphGrep Graph Database Representation

Figure 2.2 [Giugno and Shasha, 2002, Shasha et al., 2002] illustrates how a database of three graphs g_1, g_2 and g_3 is represented as a set of paths, and how the fingerprint of the database looks like. In the database, fingerprint rows contain the number of id-paths associated to each key (hash value of label-path) in each graph.

During query evaluation, the database is filtered by comparing the fingerprint of the query with that of the database. Queries are decomposed into patterns with length which is less than or equal to the predefined maximum length. Subgraphs are matched by evaluating patterns against the filtered graphs resulting in smaller search space of potentially matching graphs.

GraphGrep can be used to index message sequences in business process models as follows: (i) store message sequence paths from start such that message sequences have a predefined maximum length. The fingerprint of the database can be created similar to the approach described in this section (Figure 2.2 (e)). To evaluate a query, the query business model is partitioned into message sequences of the required length and a fingerprint of the query generated and compared against that of the database. This filters the space of message sequences that should be searched. Generating fingerprints based on message sequences enables intersection checking.

The problem with this indexing approach is that fixing the maximum message sequence length can result in false misses due to potentially infinite sequences in cyclic processes. Thus this

approach does not address the problem presented in this dissertation.

2.3.4 Set indexing

There are three common ways to index data with set-valued attributes: (i) extendible signature hashing, (ii) signature trees and (iii) inverted files
[Helmer, 1997, Helmer and Moerkotte, 2003]. Extendible signature hashing though useful for other set operations like set equality, subset and superset queries, do not support intersection operations. Inverted files are implemented based on B^+-trees, which is discussed in an earlier section. We therefore focus on signature trees.

The signature tree has two types of nodes: leaf nodes and internal or non-leaf nodes. Each internal node consists of entries of the form *(child node pointer, bounding set)*. *child node pointer* is a pointer to the root of a sub-tree where items of the root node are contained in the *bounding set*. The set containment relation between bounding sets is transitive such that each bounding set is contained in the bounding set of its parent, and this propagates from the leaf node to the root. The bounding set of a child node is computed by taking the union of all sets of its child nodes. Leaf nodes contain sets of data items and their references. The signature tree indexes set-based data by encoding the set data into binary bit fields and uses binary bit operations for evaluating set queries such as subset, superset, equality and intersection queries.

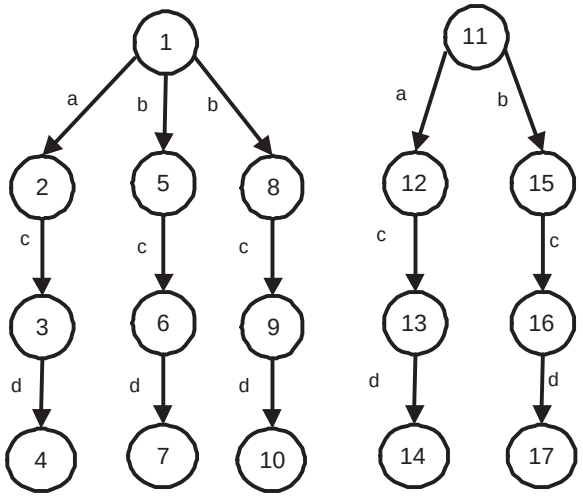
The signature tree can index business process message sets as follows: each message in the message set is encoded into a bit field of fixed length called signature length; of the signature length, exactly a fixed number of bits is set. Bit fields are superimposed using a bitwise *or* operation to yield a final signature for the set [Helmer and Moerkotte, 2003]. In a signature tree, internal or non-leaf nodes contain signatures and references to child nodes, rather than message sets themselves. Leaf nodes contain *(signature, signature reference)* pairs where the signature reference is a pointer to the business process model.

According to the performance analysis on signature trees done by Helmer
[Helmer, 1997], the signature tree supports set equality queries very well, but is poor when it comes to intersection queries. One explanation for this is the high rate of false hits inherent with intersection queries, meaning that very often, multiple branches must be traversed during a search. The poor performance and high false match rate mean that this approach is not feasible for the presented problem.

2.3.5 Semi-structured Data

Some semi-structured indexing techniques work by clustering common paths together. Examples of techniques that cluster common paths are DataGuides
[Goldman and Widom, 1997, McHugh et al., 1997, Kaushik et al., 2002] and 1-Index, 2-Index and T-index [Milo and Suciu, 1999]. DataGuides provide a dynamic and flexible way to create schemas, based on the data that is currently stored in a semi-structured database. The schema can be used for the formulation and building of queries. In addition, DataGuides can also be

used as path indexes [Goldman and Widom, 1997, Goldman et al., 1999].
The object exchange model (OEM) is used to describe semi-structured data [Nestorov et al., 1997, Buneman, 1997]: an OEM data model is a directed graph where vertices represent complex objects as well as atomic values. Edges represent the relationship between objects.



(a) Source Data (b) DataGuide

Figure 2.3: DataGuide Example

A DataGuide is defined in [Goldman and Widom, 1997] as an OEM object in which every path in a given source OEM database is represented once and only once, without additional paths not found in the source database. A variation of the DataGuide, called a strong DataGuide, has the property that if paths share the same target(s) in the DataGuide, they must also share the same target(s) in the source database. This allows paths in the DataGuide and the source database to have a one-to-one correspondence.

DataGuides can be used for indexing path expressions [Goldman and Widom, 1997]. The procedure is to compute a DataGuide from the source database using an algorithm given in [Goldman and Widom, 1997]. During DataGuide construction, a structure called *targetHash*, which is a table that maps objects in the source graph to DataGuide objects is made persistent. A target set is a set of objects reached by traversing a sequence of path labels. Another table is used to persistently store mappings from DataGuide objects to target sets. Instead of searching on the source graph, queries are evaluated against the DataGuide. DataGuide objects are mapped to their corresponding target sets in the source graph by looking up the persistent table that maps DataGuide objects to target sets. Objects in the target set are mapped back to the source graph to obtain the set of results fulfilling the query. Figure 2.3 is an example showing a source graph

and a corresponding DataGuide showing paths which have been clustered together, thus reducing the size of the graph. The example is based on the work of [Goldman and Widom, 1997], where DataGuides are introduced.

Another indexing approach is described by Milo and Suciu in [Milo and Suciu, 1999]. Milo and Suciu describe three techniques for indexing graph-based semistructured data. The indexing techniques are named 1-index, 2-index and T-index. The techniques are based on the computation of equivalence classes on nodes of a source graph. Construction of equivalence classes for a data graph is a PSPACE complete problem, so the authors used simulation and bisimulation instead, which can be computed in polynomial time [Henzinger et al., 1995, Paige and Tarjan, 1987, Buneman et al., 1997]. The cost of using simulation and bisimulation is that false matches are introduced into the index because simulation/bisimulation relations are weaker than equivalence.

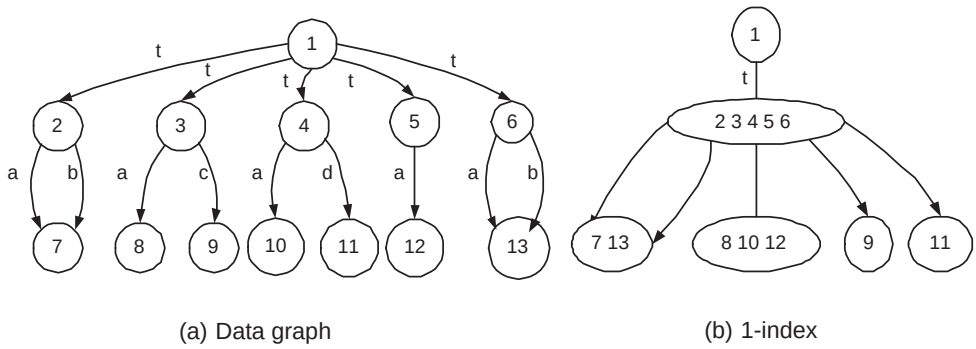


Figure 2.4: 1-index Example

The 1-index is conceptually similar to the DataGuide. Like the DataGuide, the index structure has two parts: (i) a graph whose nodes contain equivalence classes computed using either simulation or bisimulation, with edges connecting objects in each equivalence class (ii) a structure mapping objects of each equivalence class to a node in the index structure. Figure 2.4 from [Milo and Suciu, 1999] shows a data graph and its corresponding 1-index, where the number of states and transitions have been reduced by more than half in the 1-index. Like 1- and 2-indexes, the T-index is constructed based on equivalence classes using simulation or bisimulation to group equivalent objects in the data graph. Queries are evaluated against the index and unions of equivalence classes are computed to return the query result.

All the indexing approaches described in this section do not support intersection operations. There is also no mechanism to handle infinite objects corresponding to infinite message sequences (due to cycles) using the approaches. There is also no mechanism for representing and evaluating mandatory sequences.

A trie is a tree representation of a string where string characters are represented sequentially as edge labels on a tree structure from root to leaf. A Patricia trie is a compact form of a trie, with one-child nodes removed. A number is used to indicate the depth (from the root,

with the root node having 0 depth) of a node, meaning the character position to compare when searching for a key. Edges that do not differentiate between keys are therefore not represented in a Patricia trie, making it compact, and therefore saving on storage space utilization. Patricia tries grow slowly because only differences between keys are stored in the structure [Knuth, 1998, Cooper et al., 2001, Szpankowski, 1990, Morrison, 1968].

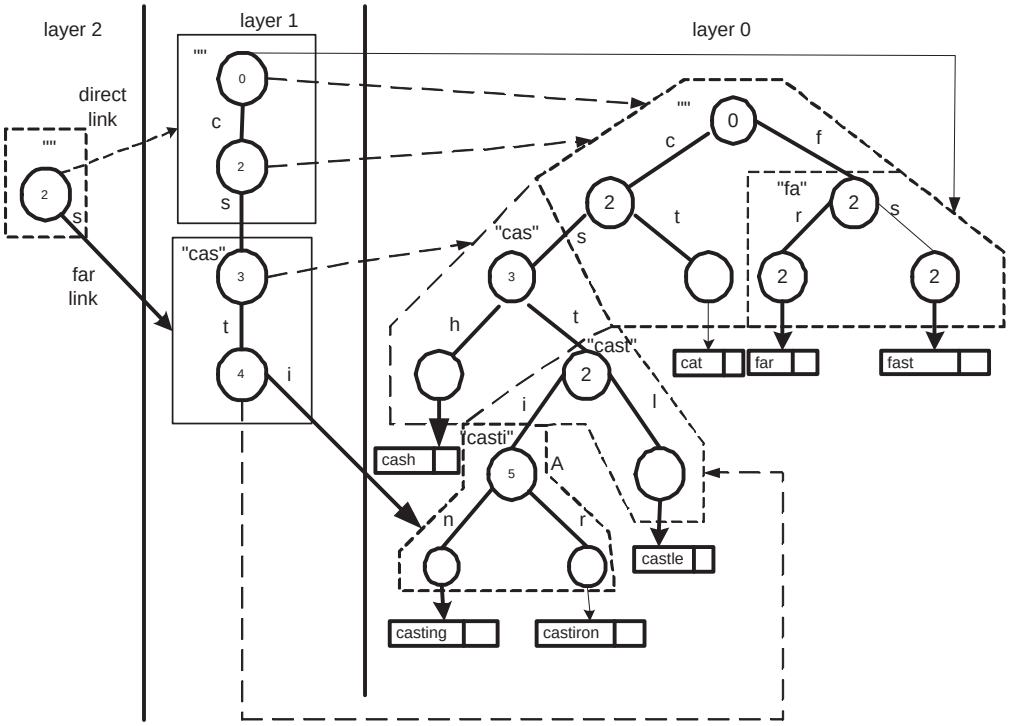


Figure 2.5: Index Fabric

Index Fabric [Cooper et al., 2001, Cooper and Shadmon, 2000] is an index mechanism based on Patricia tries [Knuth, 1998]. It introduces balancing and efficient algorithms for disk-based access to Patricia tries. The index fabric works by partitioning the Patricia trie into equal access blocks. The blocks are indexed by sub-tries that are stored in their own blocks. The effect is to introduce layers in a Patricia trie with layer 0 representing the conventional Patricia trie, layer 1, the next layer that index blocks in layer 0, and layer n indexes layer $n-1$. Patricia tries in blocks next to each other are connected using two types of links: the so called *far links* and *direct links*. Far links connect nodes in neighboring layers in a parent-child relationship, where the parent is in the higher layer, and the child is in the lower layer. Direct links connect blocks that have the same prefix, but are in different though neighboring layers, i.e., layers n and $n+1$. Figure 2.5 [Cooper et al., 2001, Cooper and Shadmon, 2000] shows the structure of the Index

Fabric index. The figure shows how a complex Patricia trie has been indexed, by dividing it into different blocks, on which a hierarchical structure is built. By following the far links (from layer 2), we can see that the sub-string "cast" is built. We can also follow a direct link from layer 2 and navigate to layer 0 where strings like "far", "fast", "cat" , etc., are built. The index has properties of a Patricia trie [Knuth, 1998] which include the ability to efficiently store long strings and adds to it the ability to give a balanced I/O to every block.

In [Cooper et al., 2001, Cooper and Shadmon, 2000], the indexing of semi-structured data using the Index Fabric technique is discussed. XML is the used data representation format. The approach is to use designators to represent tags, which are in turn stored in a dictionary. Indexing an XML document is accomplished in three steps: (i) tags are automatically generated from the XML data and stored in a dictionary (ii) root to leaf paths are encoded using the designators from the dictionary and (iii) all designator encoded paths are added to the Index Fabric.

The limitation with such indexing approaches is that they have no way to handle infinite message sequences. The intersection operation and emptiness testing are also not supported.

2.3.6 RE-trees

In [Chan et al., 2003] an indexing structure called RE-tree, for indexing regular expressions is presented. The input query to the collection of regular expressions is a string, and using the index, the collection is filtered to return a subset that matches the string input.

Like B^+-trees, RE-tree are height-balanced hierarchical structures with internal nodes that route searches and leaf nodes that contain data in the form of finite state automata (FSAs). Each leaf node carries the following information pair: an *id* and an FSA uniquely identified by the id. Each non-leaf node carries the following information: $(M_1, P_1) \cdots (M_n, P_n)$ where M_i represents an automaton and P_i, a pointer to the next-level node (like in a B^+-tree). The automaton M_i acts as a bounding automaton to the set of FSAs in the node it points to just as internal node keys in a B^+-tree bound lower level nodes. In the case of RE-trees, language containment is used to partition the search space to direct searches. Thus the principle behind the RE-tree, is very similar to that of R-trees [Guttman, 1984], RD-trees [Hellerstein and Pfeffer, 1994] and signature trees [Helmer and Moerkotte, 2003], i.e., they all rely on the notion of transitive containment relation of internal node entries to build a search hierarchical structure.

As an example (Figure 2.6 taken from [Chan et al., 2003]), the union of languages of FSAs M_6 and M_7 is a subset of the language of M_2.

Constructing the index structure is accomplished using three key operations: (i) selecting optimal node for insertion, (ii) computing the optimal bounding FSA, (iii) computing the optimal node split when overflows occur during insertions. These operations are optimized to achieve the best possible results. For example: from a set of FSAs in a node, the optimal node is selected in order to minimize the expansion of the FSA; the optimal bounding FSA is achieved by making sure that its language is minimized, and the optimal node split when overflows occur is achieved by partitioning the node, say into two, such that the union of their languages is minimum [Chan et al., 2003].

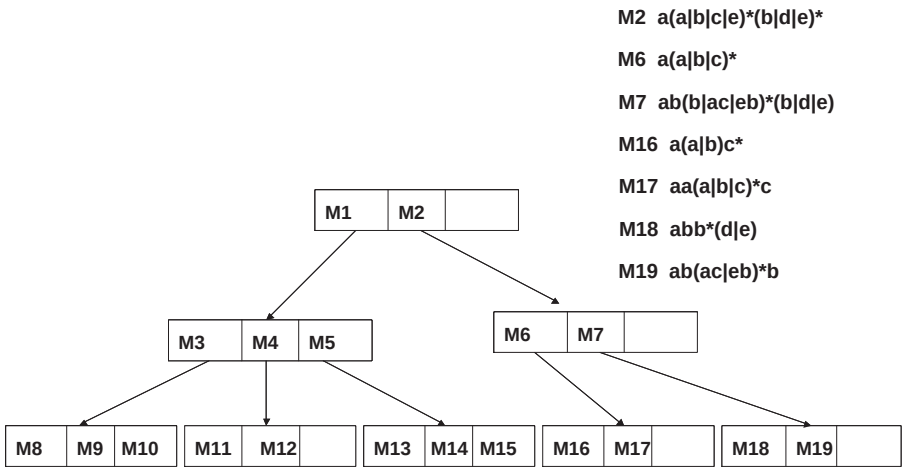


Figure 2.6: RE-tree Structure

The problem how to compute a bounding automaton with a limited number of states where the bounding automaton is minimized is NP-Hard [Chan et al., 2003]. The authors therefore used heuristics to overcome this. The problem of computing an optimal node split during overflows in order to minimize the union of the languages of the resulting FSA is also NP-Hard. So heuristics are again used.

The indexing approach does not have a mechanism to express and handle mandatory message sequences. In addition, the complexity of searching based on intersection is very high. In the worst case, every path must be traversed, and at each node, intersection (which takes quadratic time) must be performed to determine which path to follow.

Table 2.3: Indexing Techniques Assessment

Property\ Index Technique	Finite Language	Infinite Language	Intersection Operation	Query Eval Complexity	Number of False Matches
B^+-tree, Hashing	✓	×	×	N/A	N/A
OODB indexes	✓	×	×	N/A	N/A
GraphGrep	×	✓	×	N/A	N/A
DataGuides	✓	×	×	N/A	N/A
1-2-T-Index	✓	×	×	N/A	N/A
Signature Trees (Set Indexing)	✓	✓	✓	High	High
Patricia trie	✓	×	×	N/A	N/A
RE-trees	✓	✓	✓	High	None

2.3.7 Assessment of Indexing Techniques

Table 2.3 summarizes the indexing techniques presented in this sub-section. The checkmark, N A, and cross symbols have the usual meaning as described in Section 2.2 and Section 2.1. From the table, all indexing techniques are able to express finite languages, but only two (Signature trees and RE-trees) are capable of expressing infinite languages as can occur in business process specifications. The two techniques also support intersection queries. However, both Signature trees and RE-trees exhibit poor query evaluation performance for intersection-types of queries. The poor performance is due to the high computational complexity for evaluating intersection queries as already described in the previous sub-sections. In addition, signature trees exhibit a high number of false matches due to the information lost during the encoding of messages to fixed-length bit vectors. A more efficient technique for indexing infinite languages using intersection operations is therefore needed.

3 Matchmaking of Business Processes

This chapter presents a formal model for representing business processes for bilateral matchmaking. The chapter begins by presenting the minimum set of requirements the model must fulfill. The set of requirements is used as a basis for formal model development. A formal model is described and relevant definitions needed to understand the model are presented formally. Examples are used throughout the chapter to illustrate the application of the formal model.

3.1 Model Requirements

The formal model must fulfill a minimum set of requirements in accordance with the problem statement presented earlier in the dissertation. As a reminder, the functional requirements presented in Chapter 1 include the following (i) representing (potentially infinite) message sequences that are to be compared for deciding matches (ii) intersection operations support (iii) deciding emptiness of intersected model and (iv) expressing mandatory message sequences. Support for the stated requirements will allow us to compute matching business processes. In the following subsections, the basic requirements for the formal model will be described.

3.1.1 Represent (Potentially Infinite) Message Sequences

The examples presented in Chapter 1 showed that business process specifications can contain cyclic descriptions of message sequences. For example the buyer process sending a purchase order status request message to the seller and getting back a response for each request on a regular basis is a cyclic process as illustrated in Figure 1.2 and Figure 1.3. If any part of the business process description contains cyclic message sequences, that business process will contain an infinite number of message sequences, as a result of the cyclic part. As a corollary, message sequence lengths of cyclic parts of the business process are infinite. Thus the model must be able to represent potentially infinite message sequences as a basic requirement.

3.1.2 Intersection Operation Support

From the problem statement, intersection is one of the operations to be performed. It follows that the model must support intersection of message sequences. Since intersection is the central operation, it is important that whatever model is used, intersection will be decidable and be computable in polynomial time, otherwise the model is of no practical use, e.g., if the complexity is too high.

3.1.3 Decision Problem of Emptiness/ Non-emptiness

Having computed intersection, the decision problem of determining whether the set of message sequences resulting from the intersection model is empty or not, must be solved. If a non-empty set results, then the conclusion is that the two business processes match, otherwise they do not match. Thus the formal model must have mechanisms to handle this decision problem in polynomial time.

3.1.4 Express Mandatory Message Sequences

There is also need to be able to express mandatory message sequences within the business process description. For this we need to consider how to represent this within the business description. Thus the formal model must be able to handle mandatory parts of the business process description as well.

3.1.5 Other Requirements

There are other requirements to be considered when selecting a model. Computational and storage complexities are important in order to utilize the results of this work in practical environments. The other issue to consider is support for basic constructs that are necessary for building business processes. Examples are the *sequence* construct, which allows the construction of message sequences, the *iteration* or *looping* construct which allows the construction of loops or cycles in business processes, *selection* or *branching* which is needed to choose between multiple execution paths and constructs for expressing conditions. We considered these three, as basic constructs that are necessary to construct our business processes for matchmaking. There are also constructs we considered desirable, but not essential in our problem domain. An example is *parallelism* which comprises an *and-split* and an *and-join* and will be used for monitoring parallel running processes, thus is useful for executing processes that run in parallel, albeit with higher complexity. The parallelism construction can partly be simulated by enumerating the different branches resulting in lower computational complexity.

The ability to express parallelism is not critical in our application scenario. This is because in business process matchmaking, we are interested in whether certain message sequences are fulfilled by a partner or not, irrespective of whether they execute in parallel with other sequences or not.

3.2 Formal Model

Our goal is to find a model that can be used for indexing to support matchmaking queries. Thus it is important that the selected model has at least a feasible complexity. From the analysis carried out in Chapter 2, the model with the most feasible complexity is the finite state automata, where both intersection and non-emptiness operations can be carried out in polynomial time. FSAs do

Table 3.1: A Subset of RosettaNet Messages

acronym	Message name	PIP
p	Purchase Order Request	PIP3A4
p'	Purchase Order Confirmation	PIP3A4
s	Purchase Order Status Query	PIP3A5
s'	Purchase Order Status Response	PIP3A5
b	Billing Statement Notification	PIP3C5
r	Remittance Advice Notification	PIP3C6
n	Advance Shipment Notification	PIP3B2
c	Purchase Order Cancellation Request	PIP3A9
c'	Purchase Order Cancellation Confirmation	PIP3A9

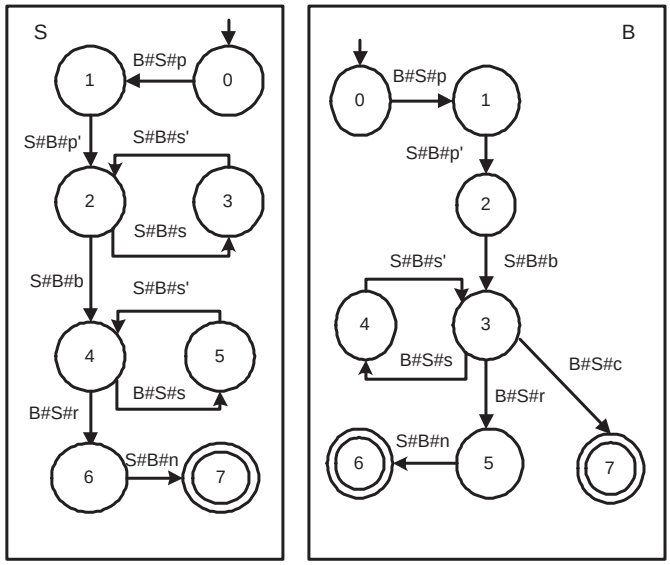
not have parallel execution semantics, as provided by more expressive approaches like Petri Nets [Peterson, 1981]. However parallelism can be simulated by enumerating the different branches, resulting in the same expressiveness as FSAs. We will extend finite state automata (FSAs) to handle mandatory message sequences. FSAs are extended with annotations to model mandatory parts of a business process description. In the following sub-sections, FSAs and their operational semantics will be described, followed by a description of their extension.

3.2.1 Finite State Automata (FSAs)

FSAs can be used to represent sets of message sequences (possibly infinite)
[Hopcroft et al., 2001]. FSA states can represent the states of a business process, where the FSA start state represents the business process start state and final states denote final business states. FSA transitions represent a change in the state of a business process, triggered by incoming or outgoing messages or other internal events of a business process.

FSAs are closed under the intersection operation, which is performed in quadratic time on the number of states and transitions. There also exists a linear time algorithm to decide emptiness. Basic constructs such as sequence and iteration or looping are supported by the model. Two business processes match if the language of their intersection automaton is non-empty. Intersection is computed using standard automata algorithms as given in [Hopcroft et al., 2001]. Emptiness test computation on the intersection automaton is also computed using automata algorithms.

Table 3.1 shows a subset of RosettaNet messages, their PIPs and message acronyms as used in examples throughout this chapter. Figure 3.1 is an extension of the example given in the motivation section (Figure 1.2) for the seller S and buyer B processes. In the figure, nodes represent states of a business process; the end states are identified by a double circle. The start state has a node with an incoming edge that has no source state. Edges represent state transitions, which are labeled with messages denoted as *sender#receiver#messageName*, where *sender* represents the role that sends the message, *receiver* represents the recipient role within the bilateral collaboration and *messageName* is the message name as already explained in the introductory



(a) Seller Process (b) Buyer Process

Figure 3.1: Seller and Buyer Processes

chapter. The roles and message names are all standardized based on existing standards such as the RosettaNet specification [RosettaNet, 2005].

The seller business process (Figure 3.1 a)) can be summarized as follows: the business process is triggered by the receipt of *B#S#p*, being a purchase order request message *p*, sent from a buyer *B* to a seller *S*. The seller responds with a message *S#B#p'*, which a purchase order confirmation message sent from the seller *S* to buyer *B*. Thereafter, one of two things can happen: the seller can send a billing statement notification (*S#B#b*) message to the buyer, or he can receive a purchase order status request (*S#B#s*) message. If the seller has sent a billing notification statement message, next he expects to get a remittance advice (*B#S#r*) message from the buyer, afterwhich he sends a shipping notification (*S#B#n*) message to notify the buyer of shipment of the order. On states 2 and 4, the buyer can check the purchase order status by sending a message *B#S#s* to the seller, and the seller responds with *S#B#s'*, updating the buyer about the purchase order status.

The standard semantics of automata is an optional execution of transitions. For example in state 2 of the seller business process, either the transition with target state 4 can be taken or that with target state 3 via transitions with labels *S#B#b* and *B#S#s* respectively. The buyer cannot express the fact that certain message sequences are mandatory using an FSA only. For example suppose the buyer is interested in matching only those sellers that support remittance advice notification and purchase order cancellation messages within their business process, after

a billing notification message. In that case the seller and buyer processes will not match because the seller does not support cancellation (see Figure 3.1). However, FSAs cannot express these semantics. According to the optional execution of transition semantics of FSAs, the seller and buyer processes in Figure 3.1 will match because the cancellation message requirement is not enforced. We can however extend traditional FSA semantics to express mandatory message sequences. Below formal definitions of FSAs and associated operations are presented. They are based on definitions from [Hopcroft et al., 2001].

Definition 3.2.1 (Finite State Automata)
A finite state automaton A is represented as a tuple, $FSA : A = (Q, \Sigma, \Delta, q_0, F, role)$ *where :*

- Q *is a finite set of states,*
- $\Sigma \subseteq R \times R \times M$ *is a finite set of messages in* M *sent by a sender in* R *to a receiver in* R*, where* R *represents the set of roles,*
- $\Delta \subseteq Q \times \Sigma \times Q$ *represents labeled transitions,*
- q_0 *a start state with* $q_0 \in Q$*,*
- $F \subseteq Q$ *a set of final states,*
- $role \subseteq R$*, the role played by the party represented by this automaton.*

In the above definition, the alphabet consists of triples of the form *sender#receiver#messageName* where *sender* is the role played by the party sending the message, *receiver* is the role played by the party that receives the message within the bilateral collaboration and *messageName* is the name of the message. Two transition labels are equivalent if they have the same sender, receiver and message name. We have also extended the standard FSA definition from [Hopcroft et al., 2001] to include role information. The role of the party modeled by the finite state automaton is represented by $role \subseteq R$, where R is a finite set of roles.

An automaton A generates a language $L(A)$ which enumerates the (possibly infinite) set of all message sequences supported by a business process. Two FSAs can be checked for intersection by checking the languages they support for common message sequences. If the language supported by the intersection automaton is non-empty, we conclude that the modeled business processes match, if the language is empty, we conclude the business processes do not match.

The operational algorithm for computing the intersection of two FSAs is derived by extending the cross product construction algorithm described in [Hopcroft et al., 2001] as described below. The resulting intersection FSA is checked for non-emptiness by traversing it from the start state until a final state is reached.

Definition 3.2.2 (Intersection of Two FSAs)
Let A_1 and A_2 be two FSAs, where
$A_1 = (Q_1, \Sigma_1, \Delta_1, q_{10}, F_1, role_1)$ *and*
$A_2 = (Q_2, \Sigma_2, \Delta_2, q_{20}, F_2, role_2)$.
The FSAs A_1 and A_2 intersect iff $role_1 \cap role_2 = \emptyset$. The intersection of A_1 and A_2 is $A = (Q, \Sigma, \Delta, q_0, F, role)$, with

- $Q = Q_1 \times Q_2$,
- $\Sigma = \Sigma_1 \cap \Sigma_2$,
- $\Delta = \left\{ \left((q_{11}, q_{21}), \alpha, (q_{12}, q_{22}) \right) \mid (q_{11}, \alpha, q_{12}) \in \Delta_1 \wedge (q_{21}, \alpha, q_{22}) \in \Delta_2 \right\}$,
- $q_0 = (q_{10}, q_{20})$,
- $F = F_1 \times F_2$ *and*
- $role = role_1 \cup role_2$.

The standard definition of FSA intersection has been extended by including the condition $role_1 \cap role_2 = \emptyset$. This condition is needed to ensure that two business processes from the same role cannot be matched. As an example, this prevents two identical business processes from the *seller* role from being matched. The principle behind the matchmaking algorithm is that only business processes from complementary roles can be matched. Thus we make the assumption that only complementary roles are involved in the matchmaking process. The side effect of this assumption is that our definition of intersection is not distributive. The lack of distributivity in our intersection definition is a result of the deliberate decision we made not to allow parties belonging to the same role to match. The lack of distributivity however does not affect bilateral matchmaking of business processes, thus does not limit our approach.

Be aware that there are other application domains like for example unifying business processes in the case of company mergers. As an example, in the case of two companies to be merged, it might be required to check if business processes from the two companies' sales departments match in order to unify the two departments. In this case, both roles will be the same, thus the condition $role_1 \cap role_2 = \emptyset$ is not needed, since the complementarity of roles is not required.

The implication of the intersection definition above is that an intersection automaton results if complementary roles are involved in the matchmaking process. The intersection automaton A can be tested for emptiness, e.g., using a FSA emptiness test algorithm described in [Hopcroft et al., 2001]. The algorithm is based on computing the reachability of states, within an automaton, starting from the start state q_0. The automaton accepts an empty language, if and only if no final state is within the set of reachable states.

A functional definition of an emptiness test is based on a recursive reachability function as given in Definition 3.2.3.

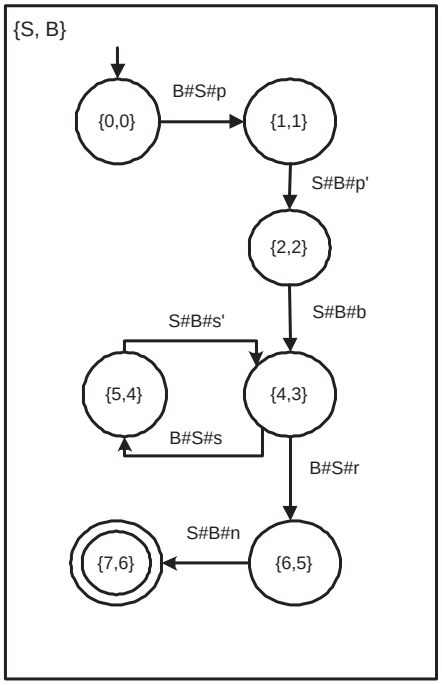


Figure 3.2: Intersection FSA of Seller and Buyer FSAs

Definition 3.2.3 (Emptiness Test)

$$Empt(curP, q_i) := \neg Reach(curP, q_i)$$

$$Reach(curP, q_i) := \begin{cases} true & if\, q_i \in F \\ \bigvee\limits_{\{q_l | \delta(q_i,l)=q_l\}} & Reach(curP.q_i, q_l) \\ & if\, q_i \notin F \wedge q_l \notin curP \\ false & otherwise \end{cases}$$

The function $Empt$ terminates, if a final state has been reached (first line of definition) or no further non-cyclic transition is available (third line). The function traverses the automaton in a deep-first manner (second line) seeking for at least one path to a final state. An automaton is empty if no final state is reachable.

The language of the intersection automaton of the seller and buyer automata (Figure 3.2) can

be tested for emptiness. This language intersection is not empty, thus the two business processes have at least one message sequence in common, meaning they match.

However, as already mentioned before, standard automata cannot express mandatory message sequences. Thus, an annotation containing this additional meta information is required for matchmaking purposes.

3.2.2 Annotated Finite State Automata (aFSA)

In this section the standard FSAs are extended to explicitly express mandatory message sequences. In this model, each state is assigned a propositional logical term.

Definition 3.2.4 (annotated FSA (aFSA))
An annotated FSA A is represented as a tuple
$A = (Q, \Sigma, \Delta, q_0, F, QA, role)$
where

- Q *is a finite set of states,*
- $\Sigma \subseteq R \times R \times M$ *is a finite set of messages in* M *sent by a sender in* R *to a receiver in* R*, where* R *represents the parties involved,*
- $\Delta \subseteq Q \times \Sigma \times Q$ *represents transitions,*
- q_0 *a start state with* $q_0 \in Q$*,*
- $F \subseteq Q$ *a set of final states,*
- $QA \subseteq Q \times E$ *is a finite relation of states and logical terms within the set* E *of propositional logic terms and*
- $role \subseteq R$*, the role played by the party represented by this automaton.*

The terms in E are standard Boolean formulas. We adapt the definition in [Chomicki and Saake, 1998] as follows:

Definition 3.2.5 (Definition of Terms)
The syntax of the supported logical formulas is given as follows:

- *the constants* $true$ *and* $false$ *are formulas,*
- *the variables* $v \in \Sigma$ *are formulas,*
- *if* ϑ *is a formula, so is* $\neg\vartheta$*,*
- *if* ϑ *and* ψ *are formulas, so is* $\vartheta \wedge \psi$ *and* $\vartheta \vee \psi$*.*

In the above definitions of terms (Definition 3.2.5), negation of a formula in *if* ϑ *is a formula, so is* $\neg\vartheta$ means that the variables are not supported. As an example $if\ (a \wedge b)$ is a formula where $a, b \in \Sigma$, $\neg(a \wedge b)$ means that the variables a, b are not supported. If this negation formula is an annotation, it means that the variables a, b cannot be taken. However, this formula has very limited practical use because if variables are not supported at a given state, transitions labeled with these variables will simply be not at this state. The standard semantics of automata is an optional execution of transitions. This is observable also in the functional emptiness test definition given above: a single path to a final state returns a *true* causing the whole disjunction to return *true* in the reachability function. Thus, the logical mapping of automata to annotated automata is an annotation containing a disjunctive expression including all transition labels as depicted in Figure 3.3. For simplicity reasons, the *OR* annotations are neglected in the automata.

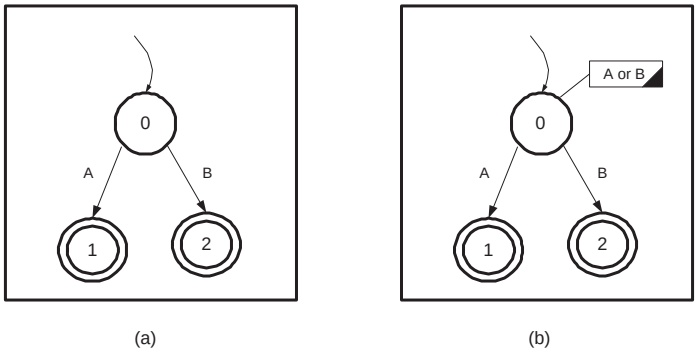
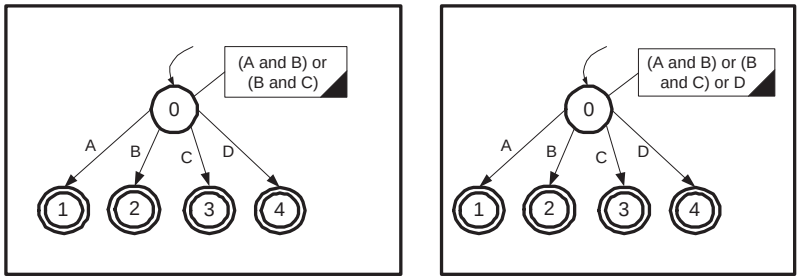


Figure 3.3: (a) Automaton (b) annotated Automaton Equivalent to a).

The definition of terms does not enforce a term to contain all labels of outgoing transitions of the associated state. Thus, annotations may be incomplete that is not containing all outgoing transition labels. Such incomplete annotations can be completed by extending them with a disjunction of all labels not contained yet. This method is best explained, if the expression is in disjunctive normal form as depicted in Figure 3.4a). The annotation means that the matching process must support message B in combination with either message A or C. Message D is unrelated to messages A, B, and C, thus represents an independent alternative which is combined with the existing term by a disjunction as depicted in Figure 3.4b).

The set of variables X^{q_i} corresponding to state q_i is defined as the set of outgoing transition labels of state q_i. This is formally expressed as:

$$X^{q_i} := \{x^{q_i} \mid (q_i, x^{q_i}, q') \in \Delta\}$$



(a) (b)

Figure 3.4: (a) Incomplete annotated Automaton (b) Completely annotated Automaton Equivalent to a).

Similar to standard Boolean logic definitions Var is the set of all variables bound in a term t^{q_i} associated to a state q_i with $(q_i, t^{q_i}) \in QA$. The formula

$$Var(t^{q_i}) \subseteq X^{q_i}$$

is not necessarily true. There might exist variables in a term associated to a state q_i without a counterpart in outgoing transition labels.

As stated above, a term t^{q_i} might be incomplete, that is

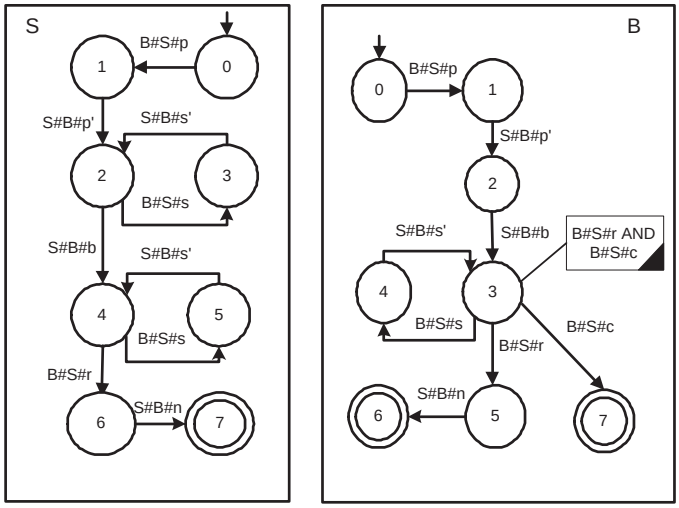
$$X^{q_i} \backslash Var(t^{q_i}) \neq \emptyset$$

and must be extended. The completed term $\tilde{t}^{q_i}$ is defined as a disjunction of the annotated term t^{q_i} associated with the state q_i and all outgoing transition labels that are not used in the term t^{q_i} so far. A formal definition is given below:

$$\tilde{t}^{q_i} := t^{q_i} \vee \Big(\bigvee_{x \in X^{q_i} \backslash Var(t^{q_i})} x \Big)$$

3.2.3 Annotated FSA Example

Suppose the buyer in Figure 3.1 b) wants to insist that matching sellers must support both cancellation and remittance messages after the billing notification message has been received.



(a) Seller Process (b) Buyer Process

Figure 3.5: Seller and Buyer Process with Logical Annotations

To express this, we include a logical term *B#S#r* AND *B#S#c* on state 3 of the automaton representing the buyer. This is shown in Figure 3.5.

3.2.4 Intersection of aFSAs

Matchmaking business processes has been defined as the non-empty intersection of automata, representing business processes. The intersection automaton of two automata contains the language accepted by both automata. Therefore, the annotation of the result automaton must support the annotation of the first AND the annotation of the second automaton.

The aFSA intersection definition is formally described below:

Definition 3.2.6 (Intersection of Two aFSAs)
Let $A_1 = (Q_1, \Sigma_1, \Delta_1, q_{10}, F_1, QA_1, role_1)$, *and*
$A_2 = (Q_2, \Sigma_2, \Delta_2, q_{20}, F_2, QA_2, role_2)$ *be two aFSAs.*
aFSAs A_1 *and* A_2 *intersect iff* $role_1 \cap role_2 = \emptyset$. *The intersection aFSA* $A = A_1 \cap A_2$ *is*
$A = (Q, \Sigma, \Delta, q_0, F, QA, role)$, *with*

- $Q = Q_1 \times Q_2$,
- $\Sigma = \Sigma_1 \cap \Sigma_2$
- $\Delta = \left\{ \left((q_{11}, q_{21}), \alpha, (q_{12}, q_{22}) \right) \mid (q_{11}, \alpha, q_{12}) \in \Delta_1 \wedge (q_{21}, \alpha, q_{22}) \in \Delta_2 \right\}$,

- $q_0 = (q_{10}, q_{20})$,
- $F = F_1 \times F_2$ *and*
- $role = role_1 \cup role_2$.

$$QA = \bigcup_{\substack{q_1 \in Q_1, \\ q_2 \in Q_2}} \begin{cases} \{((q_1,q_2), e_1 \wedge e_2)\} & if (q_1,e_1) \in QA_1, (q_2,e_2) \in QA_2 \\ \{((q_1,q_2), e_1)\} & if (q_1,e_1) \in QA_1, q_2 \in Q_2, \\ & \quad \nexists e'.(q_2,e') \in QA_2 \\ \{((q_1,q_2), e_2)\} & if (q_2,e_2) \in QA_2, q_1 \in Q_1, \\ & \quad \nexists e'.(q_1,e') \in QA_1 \\ \emptyset & otherwise \end{cases}$$

The intersection definition above is a slight extension of the automaton intersection definition given in Definition 3.2.2. The condition $role_1 \cap role_2 = \emptyset$ is still needed to ensure that two business processes from the same role cannot be matched. We again make the same assumption as discussed before for finite state automaton intersection that the querying role and the role to be matched from the repository are different. This assumption is valid because the goal of service matching is to find which complementary roles are able to fulfill a bilateral collaboration to realize a business task such as purchase order and fulfillment.

The main extension to the FSA intersection definition is the inclusion of annotations. The annotations are maintained independent of the automaton structure, with states being annotated. The evaluation of the resulting annotated automaton with regard to matchmaking is done during emptiness testing as is explained below.

The intersection annotated automaton of the seller and buyer aFSAs of Figure 3.5 is shown in Figure 3.6. The only logical expression is represented in the intersection aFSA on state 3. This annotation comes from state 3 of the buyer aFSA. According to the definition of aFSA intersection, this expression is added to QA, through the union operator at the state $\{4,3\}$ of the intersection aFSA. The intersection aFSA must now be evaluated for emptiness.

3.2.5 Emptiness Test of annotated FSA

The emptiness test algorithms of annotated FSAs have been published in [Wombacher et al., 2004a]. This section gives an overview of the emptiness test and a description how annotated terms are evaluated. The evaluation of annotated terms is done in accordance to standard logical interpretation for example, as defined in [Chomicki and Saake, 1998] where an interpretation is based on a valuation ν of variables. A variable is evaluated as *true* if and only if the target state of the transition labeled with the variable name can reach a final state. Thus, the word associated with the current state concatenated with the variable name is a prefix of at least one word accepted by the language of the automaton.

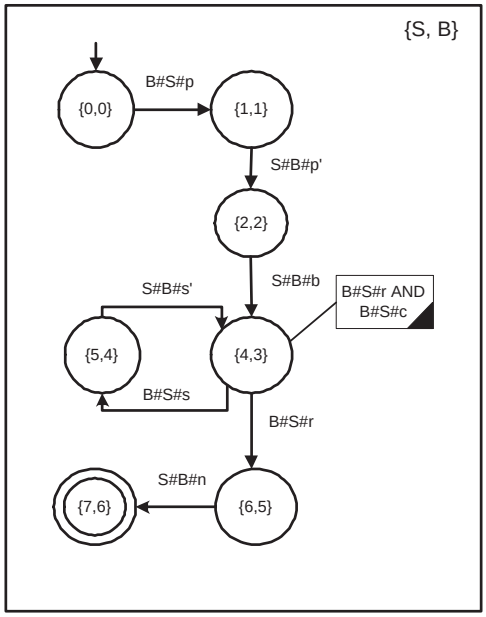


Figure 3.6: Intersection aFSA of Seller and Buyer aFSAs with Mandatory Parts

Based on this definition of truth of the annotated terms, it is required to first determine whether the target state of outgoing transitions of a state can reach a final state before evaluating the annotated term. This may result in cyclic dependencies, like for example observable in a self-loop, where the truth value of a state can not be determined, because the result depends on its own (not yet defined) truth value. This issue can be resolved by using a three-valued logic providing the standard truth values *true* and *false*, and in addition a value *indeterminate* used in case of recursion. The formal definition of the emptiness test is based on the Kleene's system of "strong connectives" [Panti, 1998] [1]. The corresponding operations of the three valued logic are negation $\neg_3$, disjunction $\vee_3$, and conjunction $\wedge_3$. The corresponding truth tables are given below.

	$\neg_3$
f	t
t	f
i	i

$\vee_3$	f	t	i
f	f	t	i
t	t	t	t
i	i	t	i

$\wedge_3$	f	t	i
f	f	f	f
t	f	t	i
i	f	i	i

The standard interpretation $\|.\|$ of the logic is based on the operations defined above, but must consider the current path *curP* of the evaluation to enable circle detection. The characters t, t_1, t_2,

[1] The special definition of implications of this system is not required in the presented approach

and c, and x represent terms, constant, and variable symbols respectively.

$$\begin{array}{ll} \|\neg t\|^{\mathrm{v}}_{curP} & := \neg_3 \|t\|^{\mathrm{v}}_{curP} \\ \|t_1 \vee t_2\|^{\mathrm{v}}_{curP} & := \|t_1\|^{\mathrm{v}}_{curP} \vee_3 \|t_2\|^{\mathrm{v}}_{curP} \\ \|t_1 \wedge t_2\|^{\mathrm{v}}_{curP} & := \|t_1\|^{\mathrm{v}}_{curP} \wedge_3 \|t_2\|^{\mathrm{v}}_{curP} \\ \|true\|^{\mathrm{v}}_{curP} & := t; \|false\|^{\mathrm{v}}_{curP} := f \\ \|x\|^{\mathrm{v}}_{curP} & := v_I \quad with \quad v_I \in \{t,f,i\}; x \in \Sigma \end{array}$$

The truth values of variables are derived by checking whether there exists a path to a final state starting from the current path $curP$ extended by the current state q_i and following the transition labeled with the name of the variable $x_j^{q_i}$. The value $intermediate\ i$ is returned if the transition labeled $x_j^{q_i}$ has a target state contained in the current path $curP$ concatenated with the current state q_i. This is, because the evaluation of the variable $x_j^{q_i}$ dependents on its own evaluation. In case the target state of the transition labeled $x_j^{q_i}$ is not contained in the current path nor in the current state q_i, the evaluation of $x_j^{q_i}$ is done by a function called reach $R()$ checking the reachability of a final state. In case no transition labeled with $x_j^{q_i}$ exists the evaluation is $false$ f. The formal definition of the valuation of variables is given below:

$$\|x_j^{q_i}\|^{\mathrm{v}}_{curP} := \begin{cases} i & if \Delta(q_i, x_j^{q_i}, q') \wedge q' \in curP.q_i \\ R(curP.q_i, q') & if \Delta(q_i, x_j^{q_i}, q') \wedge q' \notin curP.q_i \\ f & otherwise \end{cases}$$

Based on this valuation definition, emptiness in annotated automata denoted as $Empt'()$ is $false$ iff the reachability function $R()$ returns truth value t. Emptiness is defined by a comparison to ensure a Boolean result rather than a three-value logical result.

$$\begin{array}{rl} Empt'(curP, q_i) & := R(curP, q_i) \neq t \\ R(curP, q_i) & := \begin{cases} t & if q_i \in F \\ \|\tilde{t}^{q_i}\|^{\mathrm{v}}_{curP} & otherwise \end{cases} \end{array}$$

The reachability function $R()$ terminates with $true$ t if the current state q_i is a final state. If the current state q_i is not a final state the completed annotation must be valuated. Based on the emptiness test definition for aFSAs, the intersection aFSA of Figure 3.6 has an empty language. This is because in State $\{4,3\}$, the annotation $B\#S\#r\ AND\ B\#S\#c$ evaluates to $false$ because there is no outgoing transition label $B\#S\#c$ leaving State $\{4,3\}$. This false result propagates to the start state, resulting in an overall $false$ evaluation result.

3.3 Summary

This chapter presented a formal model for describing business processes for matchmaking. The formal model is based on the enrichment of finite state automata models with annotations associated to states. The annotations are propositional logical expressions representing outgoing transitions of finite state automata states. We call the enriched finite state automata, annotated finite state automata or aFSAs for short. Evaluation of annotations at a given state determine which transitions are mandatory and which are optional, determining which parts of a business process must be compulsorily fulfilled along a given path, for a match to occur. A formal definition for business process matchmaking was also described in the chapter. The definition is based on computing the intersection of aFSAs representing business process descriptions, and checking the language of the intersection aFSA for non-emptiness. Thus two business processes match if the language of the aFSA resulting from the intersection of aFSAs representing them (the two business processes being matched) is non-empty. A formal method for computing non-emptiness of annotated finite state automata based on the reachability of states is also presented.

4 Indexing of Business Processes

The previous chapter described how business processes are modeled as aFSAs for matchmaking of business process descriptions. The definition of matchmaking has been described as non-empty intersection of aFSAs representing the business processes. Intersection computation for ordinary FSAs requires quadratic computation time on the number of FSA transitions while deciding emptiness requires linear computation time on the number of states [Hopcroft et al., 2001]. The same operations must be performed for aFSAs. In addition, propositional logic expressions annotating aFSA states must be evaluated during emptiness testing of intersection aFSAs. The worst case complexity for evaluating expressions is exponential on the number of elements in the expression operation tree [Wombacher et al., 2004a]. However expressions are rather small, thus the exponential complexity is less of an issue. The total complexity for matchmaking of business processes is thus rather high. It follows that using sequential scanning, that is computing intersection and deciding emptiness on a collection of aFSAs with respect to a reference aFSA does not scale for large service repositories, which are anticipated in the Web service infrastructure. Since the computational complexity of a single query operation is more than quadratic, sequentially scanning through a large aFSAs collection and performing expensive individual query operations will not scale. The lack of scalability is a result of the need to search the entire collection sequentially and performing expensive query operations on each aFSA of the collection.

The traditional approach used to address the above problem is to use external indexes to organize data for efficient querying. However, traditional database indexes like B+-trees
[Gray and Reuter, 1993] do not directly support intersection operations on sets of sequences. The intersection and emptiness definitions of aFSAs as given in Chapter 3 are based on having at least a message sequence (or "word" in automata theory terminology) in common in both aFSAs and an evaluation of logical expressions. That means we can use a combination of message sequence equivalence and results of logical expression evaluations to find matching aFSAs. Thus by representing aFSA languages, we can check for intersection and emptiness based on message sequence equivalence and logical expression evaluations without losing the non-empty intersection semantics of aFSAs. The equivalence operator is supported by traditional indexes like B+-trees [Gray and Reuter, 1993] while logical expressions can be represented using bit vector indexes and evaluated using bit operators. However, the language represented by aFSAs is potentially infinite due to cycles in the business model specification. This means that in order to index such a language by standard techniques, we must find an abstraction of the language. Based on these principles, this chapter presents an indexing approach for efficiently matching business processes that are modeled as aFSAs.

4.1 Requirements

The indexing mechanisms developed in this dissertation are based in part on the use of abstractions to make the (potentially infinite) language of an aFSA finite. One way to make aFSA languages finite is to resolve cycles. Cycles are resolved using some abstraction mechanism. An abstraction causes information loss. The information loss might result in two scenarios: (i) false matches can result (ii) false misses can result due to the information loss introduced by the abstraction. This section describes requirements for indexing mechanisms for the matchmaking of business process descriptions. The requirements will act as a guide during the development of the indexing mechanisms. The indexing mechanisms must fulfill three requirements: (i) exclude false misses (ii) minimize processing time and (iii) minimize the number of false matches as already motivated in Chapter 1.

4.1.1 Exclude False Misses

False misses are also known as "false negatives" in some literature
[Lewis and Catlett, 1994]. This corresponds to information items that are missed, for example, by a search engine during a search operation, yet they are available. This can happen for a variety of reasons. As an example, in statistically-based key-word based searches, use of synonymous or hyponymous search terms can cause false misses [Horrocks et al., 2003]. In the dissertation, we impose a hard requirement to guarantee that if a match exists in the repository, it will be found. This ensures that the indexing mechanisms are of practical usefulness as service finders are guaranteed to find matches, when they exist.

4.1.2 Minimize Processing Time

The main goal of an index is to speed-up search operations in large repositories or databases, where sequential scanning could otherwise take too long. Thus the proposed indexing mechanisms must show significant performance improvements when compared to sequential scanning to justify their use. This is especially true for large repositories or databases.

4.1.3 Minimize the Number of False Matches

The abstraction causes information loss, thus false matches are possible. However, the rate of occurrence of such false matches must be kept to a minimum, for the index to be useful.

4.2 Query Specification

We have defined business process matchmaking as non-empty intersection of aFSAs. This matchmaking definition can be restated as non-empty intersection of aFSA languages. The aFSA languages are sets of message sequences or words (in automata theory terms). In this

dissertation, we treat the business process matchmaking problem as a search problem. There is a query part, being the business process of a service finder, i.e., the party searching for a matching service, and the database is a collection of business processes to be matched. These business processes belong to service providers, who play a complementary role to service finders. An example is that of a buyer and seller trying to match each other's business process.

If the query is an aFSA A, and the database is a collection D of aFSAs with $D := \{A_1, \ldots, A_N\}$, the query evaluation result is a subset R of the collection D containing all aFSAs A_i with a non-empty intersection with A, thus
$R := \{A_i \in D | L(A_i) \cap L(A) \neq \emptyset\}$, where $L(A_i)$ and $L(A)$ are languages of aFSAs A_i and A respectively and the intersection of two aFSAs and aFSA emptiness computation are defined in Section 3.2.4.

4.3 Finite Representation of aFSAs

The index representing a collection of aFSAs is constructed based on message sequences and propositional logic expressions. We have already alluded to the fact that the number of message sequences can be infinite due to cycles in business process specifications. Thus a mechanism is needed to finitely represent message sequences in the repository. This section describes and analyzes mechanisms for finitely representing message sequences without losing matchmaking semantics of the original aFSA.

4.3.1 Example

Figure 4.1 is a simplified, but illustrative example showing a seller S and three buyers B, represented by their business processes that are modeled as aFSAs. Within the aFSAs, only logical expressions with conjunctions are explicitly represented. Since the semantics of a choice in FSAs is disjunction, we do not explicitly represent annotations with disjunctions, but rely on the default semantics. In finding an abstraction to make the aFSA language finite, we ignore annotations, as they do not contribute to making the language infinite. We consider them separately when evaluating aFSA emptiness as will be shown later in this chapter.

The seller and buyers are all using RosettaNet Partner Interface Processes (PIPs) as basic building blocks for their processes. The RosettaNet PIPs and messages used in the example are shown in Table 4.1. The business process of a seller can be explained as follows: the seller expects to receive a *Purchase Order Request* message $B\#S\#p$ (PIP3A4) from a buyer and he responds with a *Purchase Order Confirmation* $S\#B\#p'$ message (PIP3A4). Having confirmed the purchase order, the seller can get *Purchase Order Status Query* $B\#S\#s$ message (PIP3A5) to which he must respond with a *Purchase Order Status Response* message $S\#B\#s'$ (PIP3A5). This query can be sent several times or not at all. Having confirmed the purchase order, the seller expects to send a *Billing Statement Notification* message $S\#B\#b$ (PIP3C5) to the buyer. In this state, the buyer may ask for purchase order cancellation through the *Request Purchase Or-*

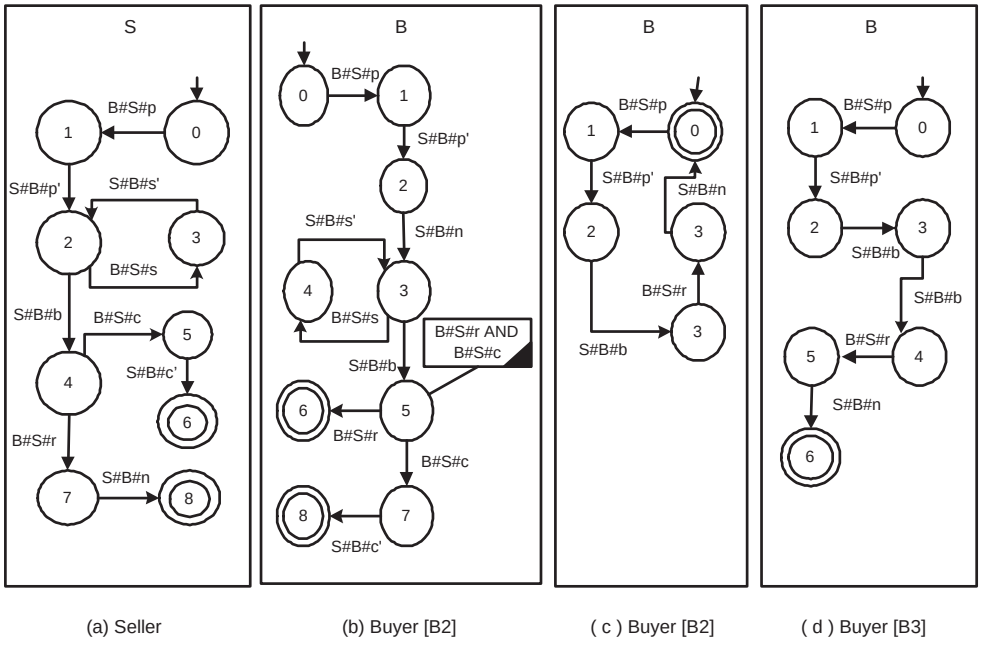


Figure 4.1: aFSAs Representing Business Processes for Seller [S] and Buyers [B1] - [B3]

der Cancellation message *B#S#c* (PIP3A9), to which the seller responds with a *Purchase Order Cancellation Confirmation S#B#c'* (PIP3A9) message. Alternatively, after receiving the *Billing Statement Notification S#B#b* message (PIP3C5), the buyer will send a *Remittance Advice Notification B#S#r* message (PIP3C6) showing his payment plan for the items ordered. Finally, the seller sends an *Advance Shipment Notification S#B#n* message (PIP3B2) informing the buyer about shipment.

4.3.2 Infinite Message Sequences

This sub-section presents possible abstractions for representing infinite message sequences. Examples described in this sub-section are based on messages already described in Section 4.3.1, except that messages may be ordered differently.

We have already stated that the number of message sequences in an aFSA is potentially infinite due to cycles in business process specifications. The abstraction to resolve cycles does not affect annotations in aFSAs since the number of annotations is always finite. Thus the abstraction does not affect the annotations which are modeled as logical expressions associated to aFSA states. The basic principle behind the abstraction to be introduced is to reduce the complexity of the potentially infinite set of message sequences accepted by an aFSA, by making the set finite. If the set is finite, the intersection operator can be replaced by simple string equivalence

Table 4.1: A Subset of RosettaNet Messages

acronym	Message name	PIP
p	Purchase Order Request	PIP3A4
p'	Purchase Order Confirmation	PIP3A4
s	Purchase Order Status Query	PIP3A5
s'	Purchase Order Status Response	PIP3A5
b	Billing Statement Notification	PIP3C5
r	Remittance Advice Notification	PIP3C6
n	Advance Shipment Notification	PIP3B2
c	Purchase Order Cancellation Request	PIP3A9
c'	Purchase Order Cancellation Confirmation	PIP3A9

of finite message sequences encoded as strings, which is readily supported by traditional index structures.

Reducing the language of a cyclic aFSA to a finite language is achieved through an abstraction ϕ which takes as input, an aFSA (potentially cyclic) and outputs a finite set of words that can be used for indexing using standard techniques. As stated in Section 4.1, we must ensure that no false misses are introduced due to the abstraction. In particular, the abstraction must guarantee that if the languages of aFSAs A_1 and A_2, denoted $L(A_1)$ and $L(A_2)$ respectively, match, the abstracted representations $\phi(L(A_1))$ and $\phi(L(A_2))$, will also have a non-empty intersection i.e., they match also. Formally, this can be denoted as:

$$L(A_1) \cap L(A_2) \neq \emptyset \longrightarrow \phi(L(A_1)) \sqcap \phi(L(A_2)) \neq \emptyset \tag{4.1}$$

Below we describe four possible abstractions to make an infinite aFSA language finite, thus indexable with standard approaches. In all the approaches, the same abstractions are applied to both the query and the data.

A1: Ignore Message Order

A possible abstraction is to ignore order in message sequences and rely on the alphabet of the aFSA i.e., individual messages for searching. Based on this approach, two business processes match if the set of messages they support have a non-empty intersection. An index can be constructed from message sets by using standard set indexing techniques such as those described in [Helmer and Moerkotte, 2003].

The problem with this approach is that it gives rise to a high rate of false matches because the order of messages is totally ignored. Another problem is that set indexing to support intersection queries is inefficient. Helmer showed in [Helmer and Moerkotte, 2003] that searching data that is set-indexed (based on signature trees) using an intersection operator is inefficient. The explanation for this was that multiple paths were being traversed along the signature tree structure

during query operations.

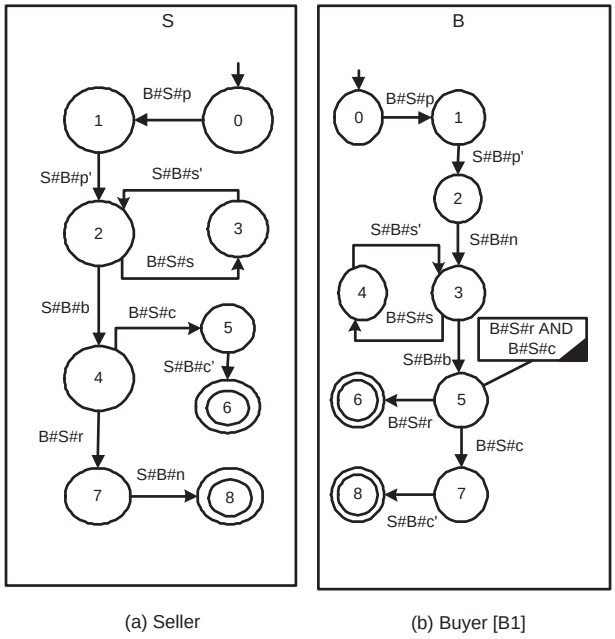


Figure 4.2: Seller and Buyer aFSAs from Seller View Point

Figure 4.2 shows the seller and buyer processes from the seller viewpoint. The difference between the seller and buyer [B1], is that the buyer expects to get shipment information *S#B#n* (PIP3B2) soon after his purchase order is confirmed by the seller via *S#B#p'* message (PIP3A4) and do remittance via *B#S#r* message (PIP3C6) later. On the other hand, the seller process sends shipping information via *S#B#n* (PIP3B2) message only after remittance advice *B#S#r* (PIP3C6) message is received. From a message level, the two processes match; however, when order is taken into account, the two business processes do not match. Thus ignoring message order and relying on message set intersection results in a false match in this case.

A2: Prune Away Cycles

Another possible abstraction is to use the aFSA, but ignoring cycles. Unlike the previous approach which ignores message order, in this approach the order of messages is not ignored. While this approach is simple and straightforward, it violates one of the stated requirements for the indexing approach. It can easily be verified that pruning away cycles from both the query and stored aFSA results in false misses, hence is not suitable for our purpose.

Consider the seller and buyer processes represented by aFSAs in Figure 4.1(a) and (c) respectively. The two processes match, since they have message sequences in common. However,

when the cycles are pruned away, the two business processes will not match. This is because the originally matching message sequences

$$\left\langle B\#S\#p, S\#B\#p', S\#B\#b, B\#S\#r, S\#B\#n \right\rangle$$

will no longer match when the back edge $(3, S\#B\#n, 0)$ is pruned away from buyer [2] process, resulting in a false miss.

A3: Remove Duplicates in Message Sequences

A third possible abstraction is to remove duplicate messages in every sequence as follows: (i) keep the first occurrence of a message in a message sequence (ii) ignore all subsequent occurrences of the same message in the same sequence.

The result of this abstraction is a new finite language where no false misses can occur during search operations, but false matches are possible. False misses are not possible because intuitively, all sequences that match originally will still match if the same amount of information is taken away from both aFSAs.

The business process for the buyer [B3] allows upto two billing statement notification messages $S\#B\#b$ (PIP3C5) to be received (see Figure 4.1). The seller and buyer $[B3]$ processes do not match, because in buyer [B3]'s business process, the purchase order confirmation message $S\#B\#p'$ (PIP3A4) is followed by two consecutive billing statement notification messages $S\#B\#b$ (PIP3C5). However, after duplicates are removed in the buyer $[B3]$ business process, the extra $S\#B\#b$ (PIP3C5) message is removed and the two business processes will match, resulting in a false match.

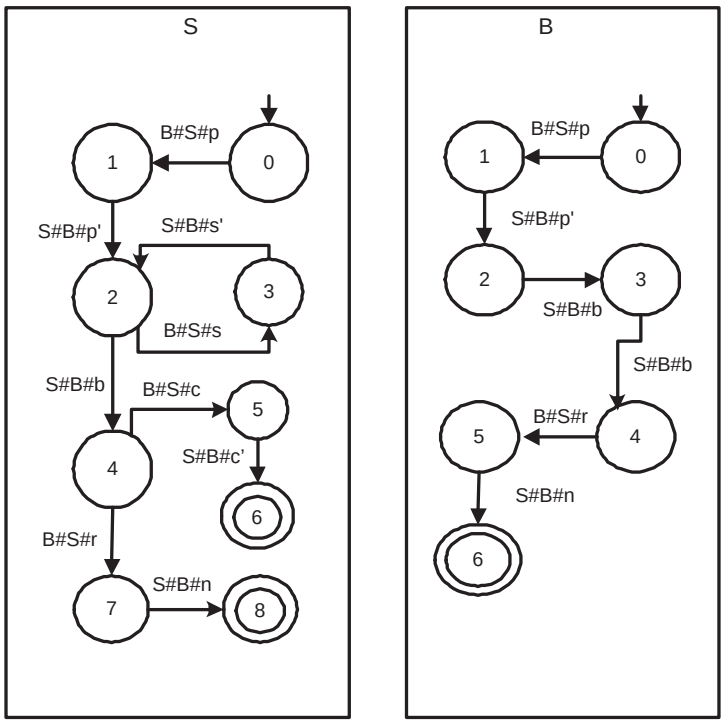
This abstraction cannot be used in its current form due to the potential high rate of false matches. However with this approach, if the same abstraction is applied to both the data and the query, no false misses can occur. Thus, although the approach has a problem of high false match rate, it meets one of the main requirements of guaranteeing no false miss.

A4: Remove Duplicates and Record Context Information through a Look-back

This abstraction builds upon the previous one, where duplicate messages are removed. The goal is to reduce the number of false matches (which is the main limitation of $A3$) by reducing the ambiguity of transitions in different aFSAs. One well known approach of resolving ambiguities in languages is to use look-ahead. Alternatively, we are using a look-back. Using look-back, we consider history information of a path traversal to make the transition information more unique. We call the history information recorded for a given position *context information* for the respective position. The context information tells us how we arrive at a given transition by looking backwards a predefined number of transitions in the history.

Like the abstraction $A3$, in Section 4.3.2, to remove duplicates, this approach allows a finite language to be generated from an infinite one and allows no false misses to occur. However,

the approach reduces the rate of false matches by introducing more precision into the language specification through the encoding of context information into the message sequence description. By increasing the amount of context information, we also increase precision, and the quality of search results.



(a) Seller (b) Buyer [B3]

Figure 4.3: Seller [S] and Buyer [B3] aFSAs from Seller View Point

Figure 4.3 shows the seller and buyer [B3] aFSAs. The false match that resulted when duplicates were removed in Figure 4.1 (d) can be avoided by using context information with a look-back of 1. In particular, the original message sequence,

$$\langle B\#S\#p, S\#B\#p', S\#B\#b, S\#B\#b, B\#S\#r, S\#B\#n \rangle$$

of the buyer [B3] business process is examined. Rather than simply considering the message itself, we now also consider the previous message, which represents the context information to be considered. Since the first message $B\#S\#p$ does not have a previous message, the \$ sign is introduced as a placeholder indicating the start of a message sequence, thus, the first element of the sequence is $[\$, B\#S\#p]$, where \$ is the context information at this position. The second

message $S\#B\#p'$ has a previous message $B\#S\#p$, thus, the second element of the sequence is $[B\#S\#p, S\#B\#p']$ with $B\#S\#p$ as context information. Following this approach, the sequence above results in

$$\Big\langle \begin{array}{l} [\$, B\#S\#p], \\ [B\#S\#p, S\#B\#p'], \\ [S\#B\#p', S\#B\#b], \\ [S\#B\#b, S\#B\#b], \\ [S\#B\#b, B\#S\#r], \\ [B\#S\#r, S\#B\#n], \\ [S\#B\#n, \#] \end{array} \Big\rangle.$$

The subsequence $[S\#B\#n, \#]$ indicates that a final state has been reached as can be ascertained from the figure. The corresponding message sequence of the seller as depicted in Figure 4.1 (a) is $\Big\langle B\#S\#p, S\#B\#p', S\#B\#b, B\#S\#r, S\#B\#n \Big\rangle$ which maps to

$$\Big\langle \begin{array}{l} [\$, B\#S\#p], \\ [B\#S\#p, S\#B\#p'], \\ [S\#B\#p', S\#B\#b], \\ [S\#B\#b, B\#S\#r], \\ [B\#S\#r, S\#B\#n], \\ [S\#B\#n, \#] \end{array} \Big\rangle.$$

Again, the subsequence $[S\#B\#n, \#]$ indicates that a final state is reached. The two sequences are not equivalent, thus they represent different message sequences. The interpretation of this is that the two aFSAs do not match. The false match has been eliminated by using a look-back of 1.

In the next sub-sections, a formal model for finitely representing message sequences based on the analysis done so far in this section will be presented. Before presenting the formal model, we will give an overview of used definitions.

4.3.3 Overview of Definitions

In this sub-section, an overview of definitions, along with their interdependencies is presented as illustrated in Figure 4.4. In Definition 4.3.1 the transformation of an aFSA to a grammar with context information, with a look-back of zero is given. This definition is used in Definition 4.3.2 to generalize the transformation of an aFSA to one with context information with a look-back of n. In Definition 4.3.3 the notion of an n-gram is introduced. We argue that the language derived from a grammar with context information is made of exactly n-gram sequences, hence we introduce the n-gram definition.

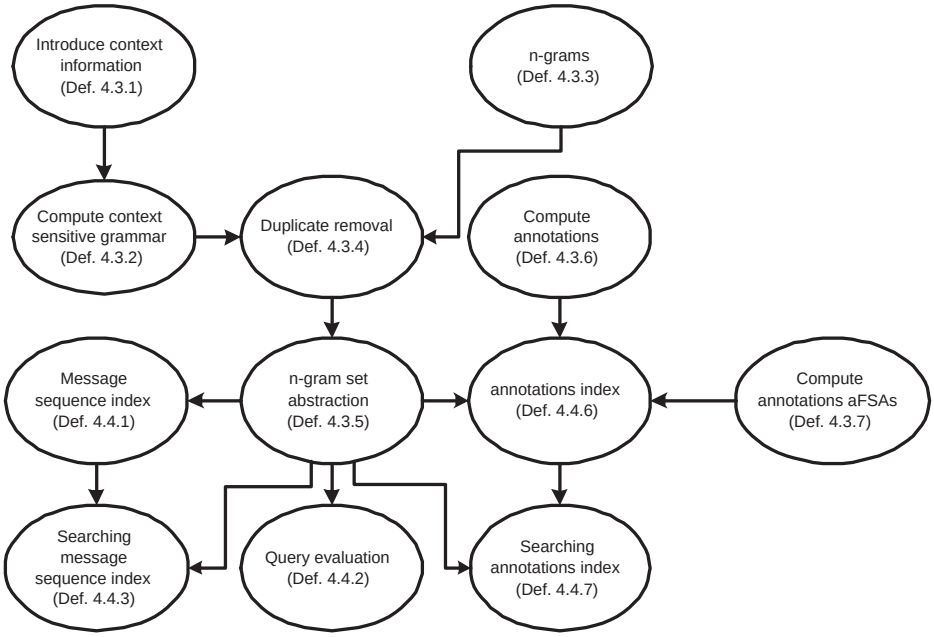


Figure 4.4: Map of Definitions

The n-gram definition is used for further formalization. Having shown the relationship between n-grams and the language derived from a grammar with context information, in Definition 4.3.4, we show how the derived language, consisting of n-gram sequences, is made finite by removing duplicates. The next definition (Definition 4.3.5) uses the finite language, as a basis to create a set-based abstraction of the finite language. The set abstraction from Definition 4.3.5 is used as a basis for several definitions, which include definitions for the message sequence index (Definitions 4.4.1), annotation index (Definition 4.4.6), query evaluation definition (Definition 4.4.2), and definitions relating to the querying of aFSAs as shown in Figure 4.4.

4.3.4 Formal Model for Finite Message Sequence Representation

This sub-section presents a formal model for finitely representing message sequences to construct an index for matching aFSAs. The approach is based on abstraction $A4$ (see Section 4.3.2), which resolves cycles by removing duplicates and recording context information through a look-back. The rest of the discussion focuses on explaining the used approach in more detail, along with the algorithms used.

Grammar Representation with Context Information

The approach is based on the construction of a grammar. A grammar is related to finite state automata in that every finite state automata can be represented by a grammar. Thus a grammar presents a different representation of a finite state automata. The use of context information in grammars is a well-known concept in formal language theory. Thus we will represent finite state automata using their grammar equivalence to facilitate their representation with context information.

The message sequences that represent an aFSA are independent of annotations, thus annotations can be independently indexed. An aFSA can thus be represented by a grammar, where annotations are ignored. The regular grammar of an aFSA can be constructed using standard techniques [Hopcroft et al., 2001]. We first describe, through a formal definition, the transformation of an aFSA to its grammar representation with context information using a look-back of 0:

Definition 4.3.1 (Introduce Context Information with Look-back of Zero)
Let $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ be an aFSA. A grammar, $G_0 = (N_0, T_0, P_0, S_0)$, with context information where look-back = 0 is derived from A such that:

- *non-terminals:* $N_0 := \Big\{[q] \mid q \in Q\Big\} \cup \Big\{[s], [s']\Big\}$, *where s and s' are newly introduced start symbols with $s, s' \notin Q$*
- *terminals:* $T_0 := \Sigma \cup \Big\{\$, \# \mid \$, \# \notin \Sigma\Big\}$, *where \$ and # represent the start and termination of a message sequence respectively*
- *start symbol:* $S_0 := [s]$
- *production rules:*

$$
\begin{aligned}
P_0 := & \bigcup_{(q_1,a,q_2)\in\Delta,\, q_2\in F} \Big\{\varepsilon[q_1] \;::=\; a, \varepsilon[q_2]; \varepsilon[q_2] ::= \#\Big\} \\
& \bigcup_{(q_1,a,q_2)\in\Delta,\, q_2\notin F} \Big\{\varepsilon[q_1] \;::=\; a, \varepsilon[q_2]\Big\} \\
& \bigcup \Big\{\varepsilon[s] \;::=\; \varepsilon, \varepsilon[s']\Big\} \\
& \bigcup \Big\{\varepsilon[s'] \;::=\; \$, \varepsilon[q_0]\Big\} \\
& \bigcup \Big\{\varepsilon[q_0] \;::=\; \#|q_o \in F\Big\}
\end{aligned}
$$

ε represents an empty string. The production $\varepsilon[q_1] ::= a, \varepsilon[q_2]$ can be expressed in short hand notation $\varepsilon[q_1] \xrightarrow{a} \varepsilon[q_2]$.

The set N_0 contains non-terminals, and is a union of aFSA states where each state is put in square brackets, and a set consisting of two additional symbols [s] and [s']. The symbols [s] and [s'] are symbols for special start productions. T_0 is a set of terminals; it is derived from the union of all input messages to the aFSA and a set of special symbols \$ and # where \$ marks the start of a word, or message sequence and # marks the end of a word or message sequence in the resulting language. S represents the grammar's start symbol, which is $[s]$, in our representation.

P_0 is the set of productions or rules. Each production is represented in Backus Naur form $\varepsilon[q_1] ::= a, \varepsilon[q_2]$ for each transition $(q_1, a, q_2) \in \Delta$. ε represents an empty string. All productions are generated from transitions, plus three special productions, not directly derived from transitions. These productions are (i) $\varepsilon[s] ::= \varepsilon, \varepsilon[s']$ (ii) $\varepsilon[s'] ::= \$, \varepsilon[q_0]$ and (iii) $\varepsilon[q_0] ::= \#$. Productions (i) and (ii) allow a \$ symbol to be used at the beginning of every sequence. The rule $\varepsilon[q_0] ::= \#$ is added only when the start state is a final state.

Productions in Definition 4.3.1 have the form $T^n \times N ::= a, T^n \times N$, where n is the look-back. The production represents a context sensitive grammar [Ginsburg, 1975]. However the form of productions is effectively modelled such that the underlying grammar is regular. This means that the context prefix on the right-hand side always starts with the full context prefix of the left-hand side, and all right-hand-sides always generate terminals to the left of a non-terminal. Next, an example to illustrate how the above definition is applied to the seller aFSA (see Figure 4.2 (a)) using a grammar with a look-back of 0 is illustrated. For the sake of brevity, role information in message specifications is omitted. The grammar for seller aFSA is G_0 with $G_0 = (N_0, T_0, P_0, S_0)$, where

- $N_0 := \Big\{[s], [s'], [0], [1], [2], [3], [4], [5], [6], [7], [8]\Big\}$,
- $T_0 := \Big\{b, n, p, p', r, s, s', c, c', \$, \#\Big\}$,
- $P_0 := \Big\{$
  $\varepsilon[s] ::= \varepsilon, \varepsilon[s']$;
  $\varepsilon[s'] ::= \$, \varepsilon[0]$;
  $\varepsilon[0] ::= p, \varepsilon[1]$;
  $\varepsilon[1] ::= p', \varepsilon[2]$;
  $\varepsilon[2] ::= s, \varepsilon[3] \mid b, \varepsilon[4]$;
  $\varepsilon[3] ::= s', \varepsilon[2]$;
  $\varepsilon[4] ::= c, \varepsilon[5] \mid r, \varepsilon[7]$;
  $\varepsilon[5] ::= c', \varepsilon[6]$;
  $\varepsilon[7] ::= n, \varepsilon[8]$;
  $\varepsilon[8] ::= \#$;
  $\varepsilon[6] ::= \#$
  $\Big\}$
- $S_0 := [s]$,

The above grammar constructs a language where each word consists of a list of comma sep-

arated groups of n + 1 characters, where $n = 0$ in this case. The resulting language is again infinite. We derive a finite language by removing duplicate character groups in the list. The duplicate character groups are removed by traversing the aFSA structure from the start state, for example via its production rules, using a depth first search algorithm. Character groups encountered along the path are recorded. Identical character groups encountered down the path of traversal are removed. This results in a finite language as shown below [1]:

$$
\left\{
\begin{array}{l}
< \$, p, p', b, r, n, \# >, \\
< \$, p, p', s, s', b, r, n, \# >, \\
< \$, p, p', b, c, c', \# >, \\
< \$, p, p', s, s', b, c, c', \# >
\end{array}
\right\}
$$

Increasing Context Information

We have described how context information is introduced into an aFSA for a look-back of 0. We have also shown how to derive message sequences from the grammar. However, to gain more precision, thus increase the quality of search results by reducing the number of false matches, it is important to use look-back values greater than one, thus increasing the context information. We now present a formal description and algorithm to increase the look-back from n to $n+1$.

Definition 4.3.2 (Computing Context Sensitive Grammar)
Let $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ be an aFSA and $G_n = (N_n, T_n, P_n, S_n)$ be a grammar representing A with a look-back of n where $n \geq 0$. A successor grammar $G_{n+1} = (N_{n+1}, T_{n+1}, P_{n+1}, S_{n+1})$ with look-back $n+1$, is computed from G_n such that:
– $N_{n+1} := N_n$
– $T_{n+1} := T_n$
– $S_{n+1} := S_n$

[1] We omit ε symbols from the language because they represent empty strings.

– *let* $\bar{a} := a_3 \cdots a_{n-1}$ *and* $\$^n[s'] ::= \$, \$^n[q_0] \in P_n$

$$
\begin{aligned}
-P_{n+1} := & \bigcup_{\substack{a_1a_2\bar{a}[q] ::= a', a_2\bar{a}a'[q_1] \in P_n \setminus \{\varepsilon[s] ::= \varepsilon, \\ \$^n[s']\}, a_2\bar{a}a'[q_1] ::= a'', \bar{a}a'a''[q_2] \in P_n}} \left\{ a_1a_2\bar{a}a'[q_1] ::= a'', a_2\bar{a}a'a''[q_2] \right\} \\
& \bigcup_{\substack{a_1a_2\bar{a}[q] ::= a', a_2\bar{a}a'[q_1] \in P_n \setminus \{\varepsilon[s] ::= \varepsilon, \\ \$^n[s']\}, a_2\bar{a}a'[q_1] ::= \# \in P_n}} \left\{ a_1a_2\bar{a}a'[q_1] ::= \# \right\} \\
& \bigcup \left\{ \varepsilon[s] ::= \varepsilon, \$^{n+1}[s'] \right\} \\
& \bigcup \left\{ \$^{n+1}[s'] ::= \$, \$^{n+1}[q_0] \right\}
\end{aligned}
$$

where $a_i, a', a'' \in T_n$.

The non-terminals, terminals and start symbol for a grammar with look-back $n+1$ are directly derived from corresponding variables in the previous grammar with look-back n, through equivalence. We will use an example to explain how productions for the grammar with look-back of $n+1$ is generated. We will use productions P_0 given in the previous example to illustrate how productions P_1 for the grammar with look-back of 1 are generated. P_0 has been given as

$$
\begin{aligned}
P_0 := \Big\{ \quad & \varepsilon[s] ::= \varepsilon, \varepsilon[s']; && (1) \\
& \varepsilon[s'] ::= \$, \varepsilon[0]; && (2) \\
& \varepsilon[0] ::= p, \varepsilon[1]; && (3) \\
& \varepsilon[1] ::= p', \varepsilon[2]; && (4) \\
& \varepsilon[2] ::= s, \varepsilon[3] && (5) \\
& \varepsilon[2] ::= b, \varepsilon[4]; && (6) \\
& \varepsilon[3] ::= s', \varepsilon[2]; && (7) \\
& \varepsilon[4] ::= c, \varepsilon[5] \ ; && (8) \\
& \varepsilon[4] ::= r, \varepsilon[7]; && (9) \\
& \varepsilon[5] ::= c', \varepsilon[6]; && (10) \\
& \varepsilon[7] ::= n, \varepsilon[8]; && (11) \\
& \varepsilon[8] ::= \#; && (12) \\
& \varepsilon[6] ::= \# && (13) \\
\Big\}.
\end{aligned}
$$

Our goal is to show how P_1 is derived from P_0 based on Definition 4.3.2. The start production is $\varepsilon[s] ::= \varepsilon, \$[s']$ being derived from $\varepsilon[s] ::= \varepsilon, \varepsilon[s']$ (1). The production $\varepsilon[s] ::= \varepsilon, \$[s'] \in P_1$ is derived from the rule $\varepsilon[s] ::= \varepsilon, \$^{n+1}[s'] \in P_1$ in Definition 4.3.2 where $look - back = 1$ and previous look-back $n = 0$.

The next production is $\$[s'] ::= \$, \$[0]$ and is derived from $\varepsilon[s'] ::= \$, \varepsilon[0]$ (2) from the rule $\$^{n+1}[s'] ::= \$, \$^{n+1}[q_0] \in P_1$ from Definition 4.3.2 with $n = 0$ and the next look-back is 1.

Table 4.2: Deriving Productions P_1 from P_0

Productions - P_0	Productions - P_1	*label*
$\varepsilon[s] ::= \varepsilon, \varepsilon[s']$	$\varepsilon[s] ::= \varepsilon, \$[s']$	(1)
$\varepsilon[s'] ::= \$, \varepsilon[0]$	$\$[s'] ::= \$, \$[0]$	(2)
$\varepsilon[s'] ::= \$, \varepsilon[0]$, $\varepsilon[0] ::= p, \varepsilon[1]$	$\$[0] ::= p, p[1]$	(2) and (3)
$\varepsilon[0] ::= p, \varepsilon[1]$, $\varepsilon[1] ::= p', \varepsilon[2]$	$p[1] ::= p', p'[2]$	(3) and (4)
$\varepsilon[1] ::= p', \varepsilon[2]$, $\varepsilon[2] ::= s, \varepsilon[3]$	$p'[2] ::= s, s[3]$	(4) and (5)
$\varepsilon[2] ::= s, \varepsilon[3]$, $\varepsilon[3] ::= s', \varepsilon[2]$	$s[3] ::= s', s'[2]$	(5) and (7)
$\varepsilon[3] ::= s', \varepsilon[2]$, $\varepsilon[2] ::= s, \varepsilon[3]$	$s'[2] ::= s, s[3]$	(7) and (5)
$\varepsilon[1] ::= p', \varepsilon[2]$, $\varepsilon[2] ::= b, \varepsilon[4]$	$p'[2] ::= b, b[4]$	(4) and (6)
$\varepsilon[2] ::= b, \varepsilon[4]$, $\varepsilon[4] ::= c, \varepsilon[5]$	$b[4] ::= c, c[5]$	(6) and (8)
$\varepsilon[4] ::= c, \varepsilon[5]$, $\varepsilon[5] ::= c', \varepsilon[6]$	$c[5] ::= c', c'[6]$	(8) and (10)
$\varepsilon[5] ::= c', \varepsilon[6]$, $\varepsilon[6] ::= \#$	$c'[6] ::= \#$	(10) and (13)
$\varepsilon[3] ::= s', \varepsilon[2]$, $\varepsilon[2] ::= b, \varepsilon[4]$	$s'[2] ::= b, b[4]$	(7) and (6)
$\varepsilon[2] ::= b, \varepsilon[4]$, $\varepsilon[4] ::= r, \varepsilon[7]$	$b[4] ::= r, r[7]$	(6) and (9)
$\varepsilon[4] ::= r, \varepsilon[7]$, $\varepsilon[7] ::= n, \varepsilon[8]$	$r[7] ::= n, n[8]$	(9) and (11)
$\varepsilon[7] ::= n, \varepsilon[8]$, $\varepsilon[8] ::= \#$	$n[8] ::= \#$	(11) and (12)

The rest of the productions are derived by recursively traversing all productions of P_0 using a depth first search algorithm, starting from $\varepsilon[s'] ::= \$, \varepsilon[0]$. The recursion terminates when a fixpoint is reached or when there exists no more productions to traverse. During the recursion, productions of P_1 are derived from pairs of successive productions of P_0 linked by a common non-terminal, like for example $\varepsilon[s'] ::= \$, \varepsilon[0]$ and $\varepsilon[0] ::= p, \varepsilon[1]$ with $\varepsilon[0]$ as the common non-terminal and context.

Table 4.2 shows productions from P_0 and derived productions P_1 using the principle described in this section. The list of productions computable from P_0 is given below:

$$
\begin{aligned}
P_1 := \{ \quad & \varepsilon[s] ::= \varepsilon, \$[s']; \\
& \$[s'] ::= \$, \$[0]; \\
& \$[0] ::= p, p[1]; \\
& p[1] ::= p', p'[2]; \\
& p'[2] ::= s, s[3] \mid b, b[4]; \\
& s[3] ::= s', s'[2]; \\
& b[4] ::= c, c[5] \mid r, r[7]; \\
& s'[2] ::= s, s[3] \mid b, b[4]; \\
& c[5] ::= c', c'[6]; \\
& r[7] ::= n, n[8]; \\
& c'[6] ::= \#; \\
& n[8] ::= \# \\
\} &
\end{aligned}
$$

Figure 4.5 illustrates the graphical representation for the grammar corresponding to the above productions.

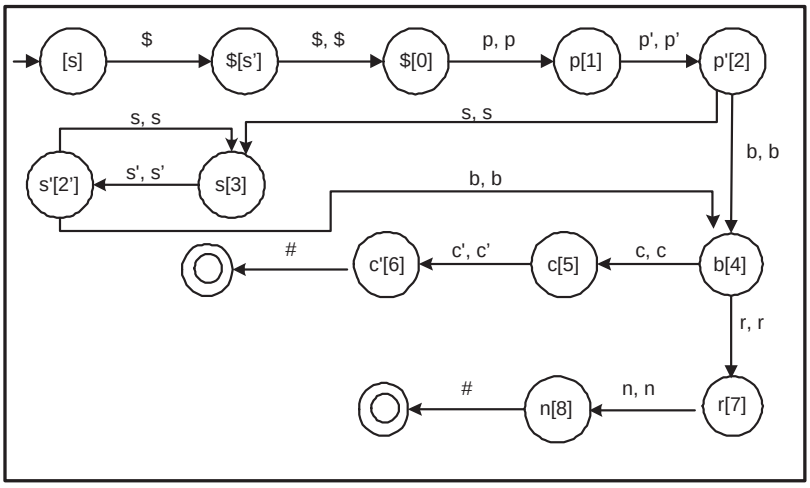


Figure 4.5: Seller Example with Look-back One

In Figure 4.5, non-terminals preceded by their context information are represented as states; transitions are labeled with terminals in accordance to the above production rules. Transition labels have the form $a, a_1 \cdots a_n a$ where a is the label for the current transition, $a_1 \cdots a_n a$ is context information for the target state reached by this transition. All transitions having # as a transition label lead into a final state.

Figure 4.5 has been presented for illustration purposes only; it is not necessary for computing the finite language. Again, the finite language is constructed from a word consisting of a list of comma separated groups of $n+1$ characters by removing duplicates of character groups in the

list. The resulting finite language from the productions of P_1 is:

$$\left\{
\begin{array}{l}
< \$\$, \$p, pp', p'b, br, rn, n\# >, \\
< \$\$, \$p, pp', p's, ss', s'b, br, rn, n\# >, \\
< \$\$, \$p, pp', p's, ss', s's, s'b, br, rn, n\# >, \\
< \$\$, \$p, pp', p'b, bc, cc', c'\# >, \\
< \$\$, \$p, pp', p's, ss', s'b, bc, cc', c'\# >, \\
< \$\$, \$p, pp', p's, ss', s's, s'b, bc, cc', c'\# >
\end{array}
\right\}$$

The productions P_1 with a look-back of one can be used to generate productions P_2 using a look-back of 2 based on Definition 4.3.2 where:

$$P_2 := \left\{
\begin{array}{l}
\varepsilon[s] ::= \varepsilon, \$\$[s']; \\
\$\$[s'] ::= \$, \$\$[0]; \\
\$\$[0] ::= p, \$p[1]; \\
\$p[1] ::= p', pp'[2]; \\
pp'[2] ::= s, p's[3] \mid b, p'b[4]; \\
p's[3] ::= s', ss'[2]; \\
ss'[2] ::= s, s's[3] \mid b, s'b[4]; \\
s's[3] ::= s', ss'[2] \mid s', ss'[2]; \\
s'b[4] ::= c, bc[5] \mid r, br[7]; \\
p'b[4] ::= c, bc[5] \mid r, br[7]; \\
bc[5] ::= c', cc'[6]; \\
br[7] ::= n, rn[8]; \\
cc'[6] ::= \#; \\
rn[8] ::= \#
\end{array}
\right\}$$

As illustrated for P_1, we can graphically represent the grammar corresponding to P_2 as shown in Figure 4.6 below. Increasing the amount of look-back results in more context information being encoded in subsequences making-up message sequences of the language. This results in a more expressive finite language because potential false matches are minimized as we have already illustrated. The quality of search results is also increased as a result of the high expressiveness of the intersecting languages. We also want to explicitly point out that both the query and data must undergo the same transformation; hence, the approach is guaranteed to find a match if one exists. The derivation of message sequences from the abstracted aFSA grammar is performed using standard techniques for deriving a language from a grammar, for example as found in [Hopcroft et al., 2001].

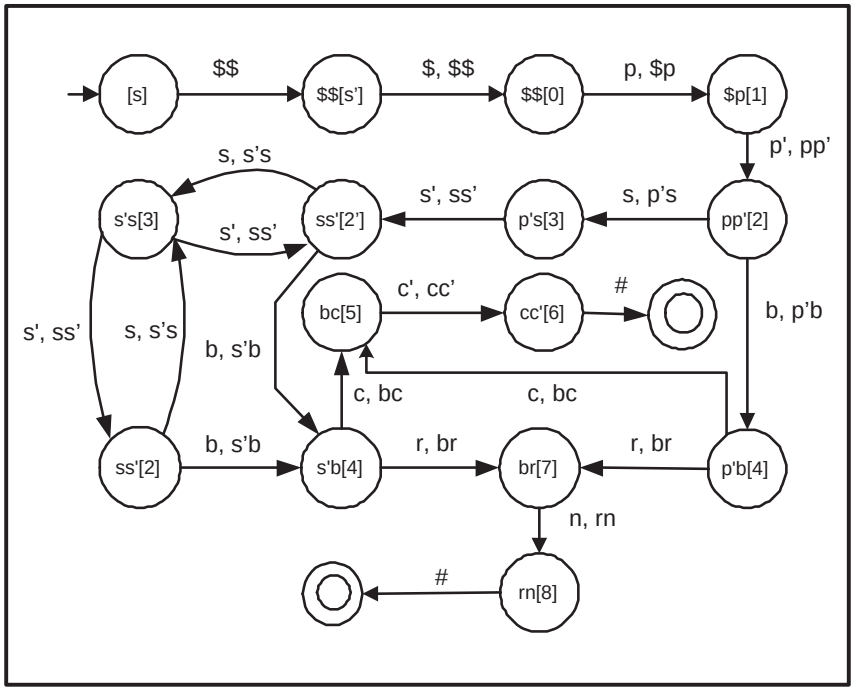


Figure 4.6: Graphical Representation of Productions P_2

N-Gram Representation

Subsequences generated by abstraction $A4$ are n-grams
[Baeza-Yates, 1992, Mahleko et al., 2005b, Kim and Shawe-Taylor, 1994]. We introduce n-grams because they will be needed for later formalization in this chapter. n-grams have been used in text indexing approaches in particular for substring matching for a long time [Baeza-Yates, 1992]. The general idea behind n-grams is representing a single long string by several strings of fixed-length n. The length n controls the complexity of all operations performed on the strings and also influences the precision of the operation result. The definition of n-grams of an aFSA language $L(A)$ for $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ is as follows:

Definition 4.3.3 (N-Grams)
Let $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ be an aFSA with language $L(A)$. A message sequence $\omega \in L(A)$ with ω represented as $\omega =< b_1 \cdots b_M >$ with $b_i \in \Sigma$, is transformed to an n-gram via function φ_n. $\varphi_n(\omega) =< \ell_1, \cdots, \ell_K >$ such that each ℓ_i is an n-gram where $\ell_i = [a_{i,1} \cdots a_{i,n}]$ for $i = 1 \cdots K$

$$a_{i,k} = \begin{cases} \$ & \text{if } i+k \leq n \\ \# & \text{if } i = K \text{ and } k = n \\ b_{k+i-n} & \text{otherwise} \end{cases}$$

where K is the number of n-grams in the message sequence and $M \leq |\Sigma|$ is the number of transition labels making-up the message sequence.

As already explained before in this section, \$ and # are special characters, which are not in the input alphabet of the aFSA. They designate the start and end of a message sequences respectively. The special character \$ is also used as a place holder for n-gram positions that are immediately unoccupied. As an example, if $< p >$ is a valid message sequence for an aFSA, it is represented by a 2-gram as $< [\$\$], [\$p], [p\#] >$. The substrings $[\$\$], [\$p]$ and $[p\#]$ are called 2-grams, where the two denotes the number of terminals used in each substring. The language derived from a grammar with context information for a look-back of n, is equivalent to a set of n-gram lists. Each n-gram list represents a message sequence. For a look-back of n, the language derived from the grammar consists of message sequences, where each message sequence is represented by a list of $(n+1)$-grams. As an example, the language of the seller aFSA derived using a look-back of one, in Figure 4.5, consists of message sequences that are made-up of 2-grams. Formally, for an aFSA A, the language derived from the grammar of A with context information based on a look-back of n, can be represented as $\Phi_n(L(A))$ where

$$\Phi_n(L(A)) := \bigcup_{\omega \in L(A)} \{\varphi_n(\omega)\} \tag{4.2}$$

The language derived from the abstracted aFSA is potentially infinite. We want to ensure that the language is finite, so that it can be used to create an index. The next definition removes duplicate n-grams from the language $\Phi_n(L(A))$ of the abstracted aFSA to make it finite, thus usable in a database index.

Definition 4.3.4 (Duplicate Removal)
Let $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ be an aFSA with language $L(A)$ and $\Phi_n(L(A)) := \{\varphi_n(\omega_1), \varphi_n(\omega_2), \varphi_n(\omega_3), \cdots\}$ with $\varphi_n(\omega_i) = \{< \ell_{i,1}, \cdots, \ell_{i,K_i} >\}$ is a potentially infinite language of A's abstracted aFSA, computed with a look-back of n; K_i is the number of n-grams in $\varphi_n(\omega_i)$. A finite language $\Psi(\Phi_n(L(A)))$ is computed from $\Phi_n(L(A))$ by removing duplicates from $\Phi_n(L(A))$, where:

$$\Psi(\Phi_n(L(A))) := \{< \ell'_{i,1}, \cdots, \ell'_{i,K'} >\}$$

with $K'_i \leq K_i$ and

$$\ell'_{i,j} := \begin{cases} \ell_{i,j} & \text{if } \nexists k < j . \ell_{i,k} = \ell_{i,j} \\ \varepsilon & \text{otherwise} \end{cases}$$

In Definition 4.3.4 above, if duplicates exist in the abstracted language $\Phi_n(L(A))$, they are removed. Operationally duplicates can be removed during traversal of the productions using a

depth first search algorithm to construct message sequences, without first creating a potentially infinite language.

N-Gram Sets

We have so far described an aFSA language abstraction based on n-gram lists. However, we would like to evaluate annotations which are represented as logical expressions associated to aFSA states. Thus we need to preserve the association between individual states and logical expressions. Without explicit n-grams associated to states, which is the case with n-gram lists, preserving the relationship between individual states and logical expressions becomes rather complicated since an n-gram list defines a complete path from start state to a final state, based on traversed transitions. Thus associating such a list, or its constituent n-grams, to individual states requires somehow creating pointers within the n-gram list to the individual states, which can result in complex structures. Apart from the complex representation, another reason why it is important to use n-gram sets (see next definition, Definition 4.3.5) as opposed to n-gram lists is to facilitate the evaluation of logical expressions at individual states, during query evaluation, because the evaluation of queries is not based on n-gram lists alone, but also on the evaluation of logical expressions too, at various states. The use of n-gram lists makes this evaluation rather complicated because at any state, the evaluation must be performed, and an n-gram list cannot be used for this evaluation without first pre-processing it to get the right prefix. The evaluation of queries based on n-gram sets will be illustrated in Section 4.4.3.

Since n-grams can be associated to states as illustrated in Figure 4.7, we can directly relate an n-gram to a logical expression. n-gram sets provide a set abstraction of an n-gram list by removing order within the list, such that individual n-grams can be stored with references to their associated logical expressions. The next definition introduces a set abstraction of a language.

Definition 4.3.5 (N-Gram Set Abstraction of a Language)
Let $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ *be an aFSA and* $\Psi(\Phi_n(L(A))) := \{< \ell'_0, \cdots, \ell'_{k'} >\}$ *is the abstracted aFSA language, with duplicates removed. The n-gram set abstraction of a language is* $\tau(\Psi(\Phi_n(L(A))))$ *where:*

$$\tau(\Psi(\Phi_n(L(A)))) := \bigcup_{<\ell'_0,\cdots,\ell'_{k'}> \in \Psi(\Phi_n(L(A)))} \bigcup_{i=0}^{k'} \{\ell'_i\}$$

Definition 4.3.5 converts the finite language obtained after the removal of duplicate n-grams, into a set of n-grams, thus ignoring the order in which individual n-grams appear. Thus the resulting language contains a set of n-grams. The n-grams are used for creating the index as will be shown later in this chapter. Note that information loss occurs due to lack of ordering introduced by sets. This information loss is not critical as each n-gram maintains local ordering within itself where the n transition labels making-up the n-gram are always ordered. In a later

section we will give a proof to show that the information loss does not result in false misses, but false matches can result.

Associating N-Grams with States

Individual n-grams are associated to aFSA states. When traversing an aFSA graph, at every state, the current n-grams related to that state can be determined. The n-grams are determined by considering the number of steps to look-back from the current state and the transition label leading to the current state. The look-back information is the context information, helping to remember previously visited states. We formally describe a function θ for mapping aFSA states to individual n-grams below.

Definition 4.3.6 (Mapping aFSA States to N-Grams)
Let $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ be an aFSA represented by a grammar $G_n = (N_n, T_n, P_n, S_n)$ using a look-back of n. n-grams derived from A via the abstraction $\tau(\Psi(\Phi_n(L(A))))$ are related to states within aFSA A by a function θ where:

$$\theta(q) := \bigcup_{a_1 a_2 \bar{a}[q'] ::= a', a_2 \bar{a} a'[q] \in P_n \setminus \{\varepsilon[s] ::= \varepsilon, \$^n[s']\}} \left\{ a_1 a_2 \bar{a} a' \right\} \tag{4.3}$$

where $q, q' \in Q$; $a_1, a_2, a' \in T_n$; $\bar{a} := a_3 \cdots a_{n-1}$; s and s' are additional states, and ε is an empty string.

Definition 4.3.6 above implies that for every production rule, we can derive an n-gram, and associate that n-gram with the target state implied by the production. The association between n-grams and states will be useful later for constructing the index for the annotations.

4.3.5 Representing Annotations in the Repository

Logical expressions representing annotations to states must be represented in the repository and evaluated during search operations. We will assume in this dissertation that logical expressions, whose expressiveness are that of propositional logic, are in disjunctive normal form (DNF). This assumption is reasonable because in logic, it can be shown that every statement can be expressed in disjunctive normal form [Mendelson, 1997]. For each disjunct of an expression in DNF, a set comprising all conjuncts is computed. As an example, for an expression in DNF such as $(a \wedge b \wedge c) \vee (b \wedge c) \vee (a)$, the sets $\{a, b, c\}$, $\{b, c\}$ and $\{a\}$ will be constructed respectively, from $(a \wedge b \wedge c)$, $(b \wedge c)$ and (a). Each set of conjuncts is treated as an annotation for an aFSA state. For example, the sets $\{a, b, c\}$, $\{b, c\}$ and $\{a\}$ represent the three annotations for the respective aFSA state. Each annotation is associated to an n-gram representing a production rule. By associating each annotation to an n-gram, and mapping this combination to a set of aFSAs, an annotation index can be constructed.

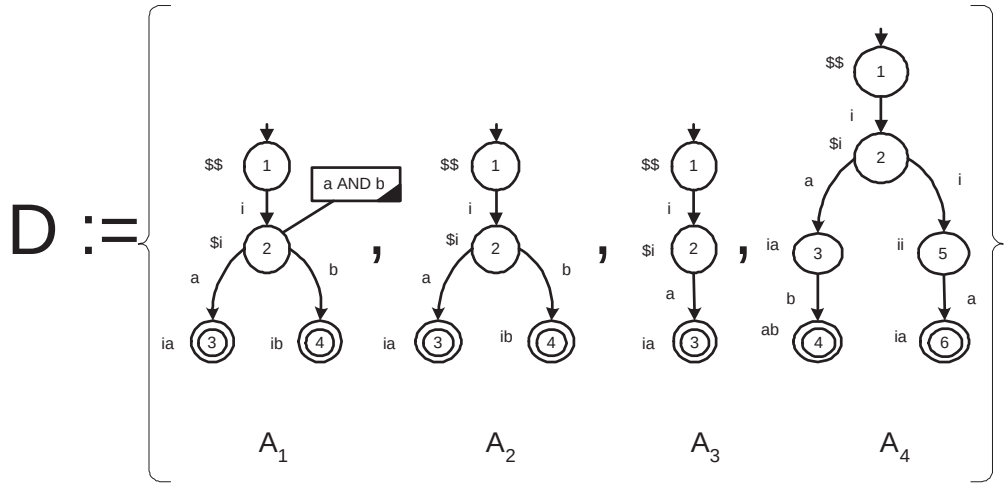


Figure 4.7: Example aFSA Collection Based on 2-grams

We first formalize the transformation of logical expressions for representation in the repository.

Definition 4.3.7 (Computing Annotations from aFSAs)
Let $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ be an aFSA where each state $q \in Q$ is completely annotated as $e := (a_{1,1} \wedge \cdots \wedge a_{1,k_1}) \vee \cdots \vee (a_{m,1} \wedge \cdots \wedge a_{m,k_m})$. The function $\rho(q)$ computes the annotations on state q where:

$$\rho(q) := \bigcup_{j=1}^{m} \Big\{ \{a_{j,1}, \cdots, a_{j,k_j}\} \Big\} \tag{4.4}$$

Figure 4.7 is an example showing a collection of four aFSAs, A_1 to A_4 where for brevity, role information has been omitted. We assume complementary roles for aFSAs to be matched throughout this example. Each aFSA has, associated with each state, a 2-gram representing a look-back of one, for each state within the aFSA. The n-grams are created based on Definition 4.3.2. As already explained, the default semantics of a choice in automata is a disjunction, so disjunctions are not explicitly represented in aFSAs. As an example, the difference between aFSA A_1 and aFSA A_2 (Figure 4.7) is that in State 2 of both aFSAs, both transition choices must be followed in A_1, while in A_2, the two transitions leaving State 2 represent optional choices.

In terms of matchmaking, aFSA A_1 matches with aFSA A_2, because all mandatory transitions of A_1 are supported by A_2, leading to finite states. However, A_1 does not match with A_3 because A_3 does not support the mandatory transition labelled with *b* at aFSA state 2 (Figure 4.7).

Table 4.3: N-Gram Table (t_1)

aFSA	2-gram (ℓ)
A_1	\$\$
A_2	\$\$
A_3	\$\$
A_4	\$\$
A_1	\$i
A_2	\$i
A_3	\$i
A_4	\$i
A_1	ia
A_2	ia
A_3	ia

aFSA	2-gram (ℓ)
A_1	ib
A_2	ib
A_4	ab
A_4	ii
A_1	a#
A_1	b#
A_2	a#
A_2	b#
A_3	a#
A_4	a#
A_4	b#
A_4	ia

Table 4.3 shows a table associating n-grams to aFSAs, where the aFSAs are given in Figure 4.7. The table comprises a mapping of aFSAs to 2-grams. Table 4.3 has been normalized such that the domain of the aFSA attribute is atomic. The table shows for example that all aFSAs in Figure 4.7 contain the 2-gram \$\$ as well as \$$i$.

Table 4.4 shows a table showing the association of annotations with aFSAs, n-grams and aFSAs where aFSAs are given in Figure 4.7. The table shows that for each n-gram, there is an annotation associated with it as described in Definition 4.3.7. This annotation is derived from the logical expression associated with that state. For example, using aFSA A_1 in Figure 4.7 as an example, state 1 is associated with annotation i, the label for the only transition leaving 1. This is represented in the annotations table of Table 4.4 by the first row. Next, state 2 of aFSA A_1 is associated with annotation a AND b which is represented as a set $\{a,b\}$. This is depicted by row five of Table 4.4. n-grams ia and ib are associated with no outgoing transitions, thus are not explicitly represented in the annotations table.

The set of annotations shown in Table 4.4 cannot be indexed directly to support set operations of intersection and complement. To index these sets, we use bit vectors. Each set in the *annotation* column of Table 4.4 is represented as a bit vector. This is achieved by first mapping the used messages to bit vectors of fixed-length. For the example in Figure 4.7, the three messages a, b and c used in the aFSAs can be mapped to bit vectors of fixed-length four, as shown in Table 4.5.

Using bit vectors in Table 4.5, the annotations in Table 4.4 can be represented as shown in Table 4.6, where the domain of the *annotation* column comprises fixed-length bit vectors, rather than sets. The length of the bit vectors can be increased if required by introducing an extra byte (or more if necessary) and re-organizing the index. The annotation $\{a,b\}$ has been mapped to a bit vector 0011 by using bitwise *or* operator (0001 *or* 0010 = 0011), where $a \mapsto 0001$ and $b \mapsto 0010$ as shown in Table 4.5.

Table 4.4: Annotations table (t_2)

aFSA	2-gram (ℓ)	annotation (g)
A_1	$$	{i }
A_2	$$	{i }
A_3	$$	{i }
A_4	$$	{i }
A_1	$i	{a, b }
A_2	$i	{a }
A_2	$i	{b }
A_3	$i	{a }
A_4	$i	{a }
A_4	$i	{i }
A_4	ia	{b }
A_4	ii	{a }

Table 4.5: Message Mapping to Bit Vectors of Fixed-Length Four

message name	bit vector
a	0001
b	0010
i	0100

Table 4.6: Annotations table (t_2)

aFSA	2-gram (ℓ)	annotation (g)
A_1	$$	0100
A_2	$$	0100
A_3	$$	0100
A_4	$$	0100
A_1	$i	0011
A_2	$i	0001
A_2	$i	0010
A_3	$i	0001
A_4	$i	0001
A_4	$i	0100
A_4	ia	0010
A_4	ii	0001

Table 4.6 can be indexed using standard indexing approaches. Queries based on bit vectors representing annotations are evaluated using efficient bitwise operators. Set intersection of annotations is evaluated using the bitwise *and* operator; set union operations are evaluated using the bitwise *or* operator while complementation is supported by negating the bit vector. There exist techniques to compress bit vectors as well as strategies to make querying based on bit vectors more efficient as described in [Wu et al., 2004]. Later we shall see how queries are evaluated using both the message sequence index and the annotation index, which is based on bit vectors.

4.3.6 False Match Analysis

False matches can be introduced by the language abstraction or due to annotations. We first discuss false matches due to the language abstraction.

False Matches due to Language Abstraction

To better understand false matches resulting from message sequences, we construct message sequences that are represented by the set of abstracted message sequences. The abstracted message sequences are derived from the original aFSA using abstraction $A4$ with a certain look-back of n. The difference between the constructed aFSA (further called the equivalence aFSA) and the original aFSA represents the set of message sequences that give rise to false matches. The construction of the equivalence aFSA is based on three factors: i) the grammar used for constructing the abstraction with a look-back of n, ii) the knowledge that duplicates are removed afterwards and iii) the fact that we are using n-gram sets and evaluation of logical expressions for matching aFSAs.

The equivalence aFSA for the seller example for the look-back from zero to two are depicted in Figure 4.8 (a) to (c) respectively. The equivalence aFSA is constructed by analyzing the language resulting from applying the abstraction as well as the production rules of the original aFSA. Note that for the sake of brevity we have omitted the sender/ receiver information on message names in Figure 4.8. The self-loops contained in the seller process with look-back zero (see Figure 4.8 (a)) indicate that due to the fact that no context information is used, every message which has been used before can be applied again. Thus, the number of message sequences represented by the set of abstracted message sequences is huge. Note that since we are using n-gram sets for matching aFSAs, every state, except the start state, is an accepting state. The self-loops of Figure 4.8 (a) disappear if the look-back is increased to one as depicted in Figure 4.8 (b), because *additional transitions like the self-loops can only be introduced in case two transitions have the same context, that is, they have the same prefix*. With regard to the example this is not the case, thus self-loops are not introduced once the look-back has been increased beyond one. Be aware that in some cases, the number of false matches can be infinite due to the cycles introduced by the additional transitions that are not contained in the original aFSA. In such cases, increasing the look-back to a bigger number will not resolve the infinite number

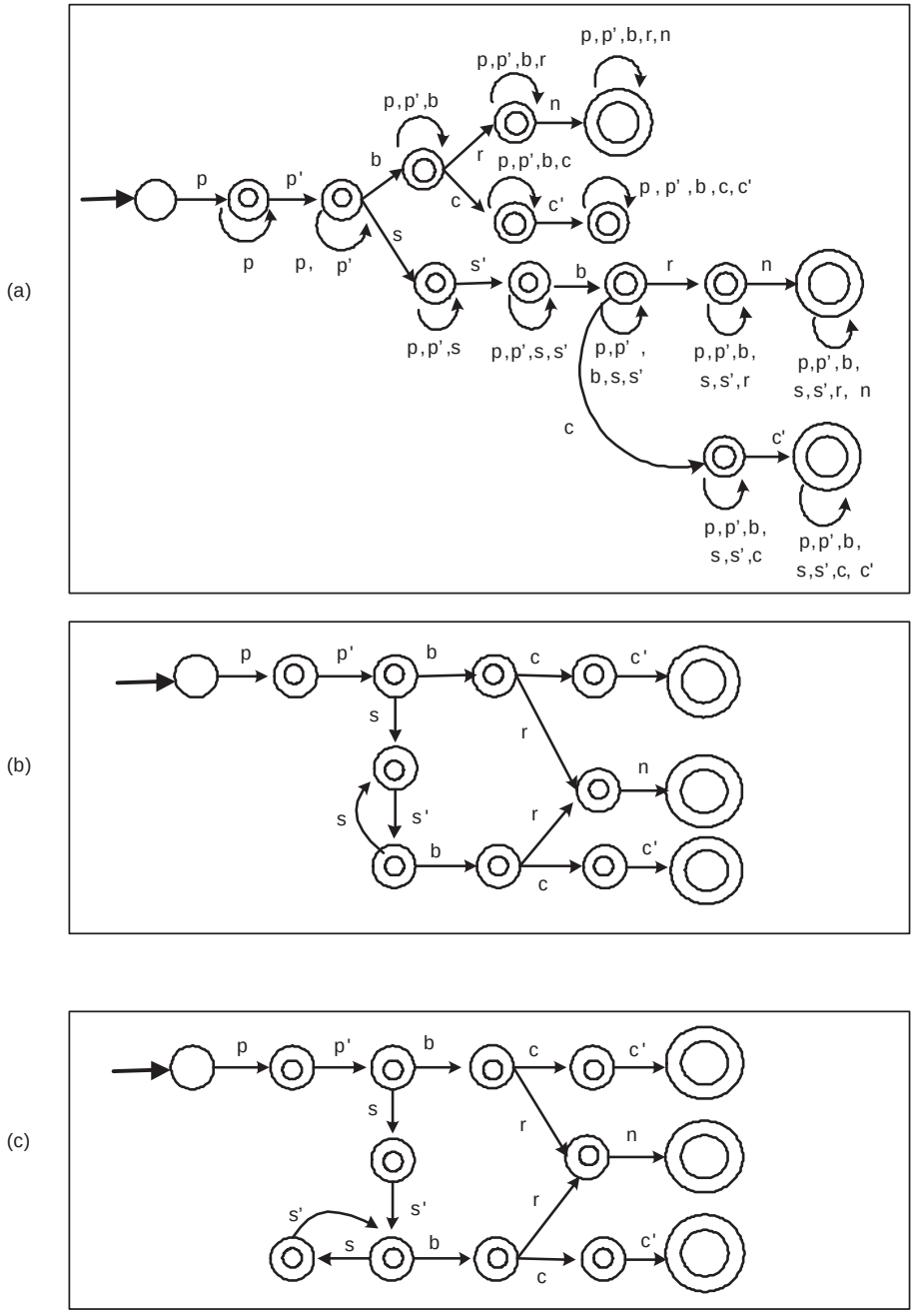


Figure 4.8: False Match Analysis with Different Look-backs n: (a) n=0 (b) n=1 (c) n=2

of false matches (see next example on Figure 4.9). However the likelihood of this occurring in practice is very rare as will be discussed in the next example.

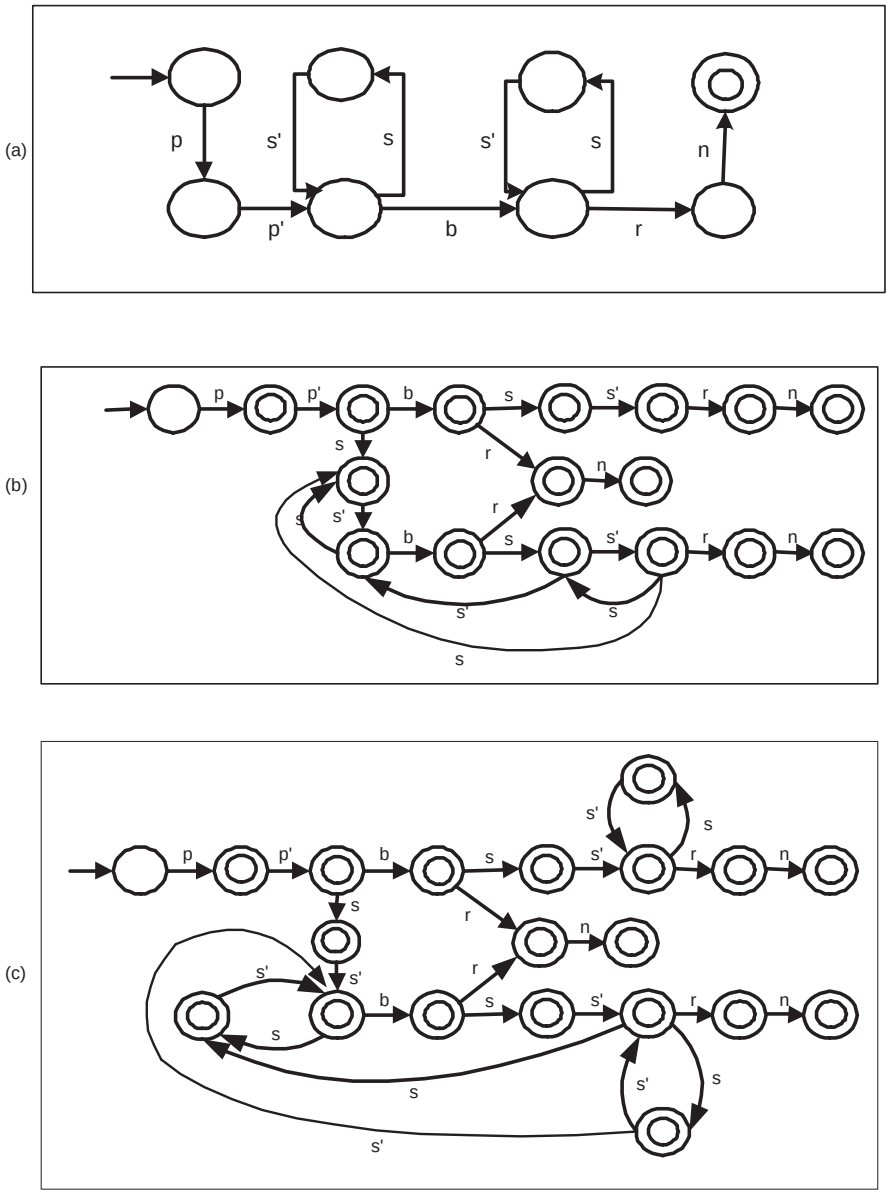


Figure 4.9: False Match Example: (a) Seller Process with Two Identical Cycles Along Path (b) n=1 (c) n=2

Figure 4.9 shows a seller business process example with two identical cycles in a single execution sequence. Additional loops have been introduced in Figure 4.9 (b) and (c). This is due to the fact that two transitions have the same context, that is, they have the same prefix in this example. The shared contexts are the two identical cycles along the same path as shown in Figure 4.9 (a). Be aware that in this case (Figure 4.9 example), the number of false matches is infinite due to the cycles introduced by the additional transitions that are not contained in the original aFSA. In this case, increasing the look-back to a bigger number will not resolve the infinite number of false matches. As can be observed for a look-back of two depicted in Figure 4.9 (c), the additional cycles are still represented. Hence, having two identical cycles in a single execution sequence always results in potentially infinite false matches. However, the occurrence of such a special case is very unlikely in the problem motivated in this dissertation.

False Matches due to Annotations

We now show that false matches can occur due to annotations.

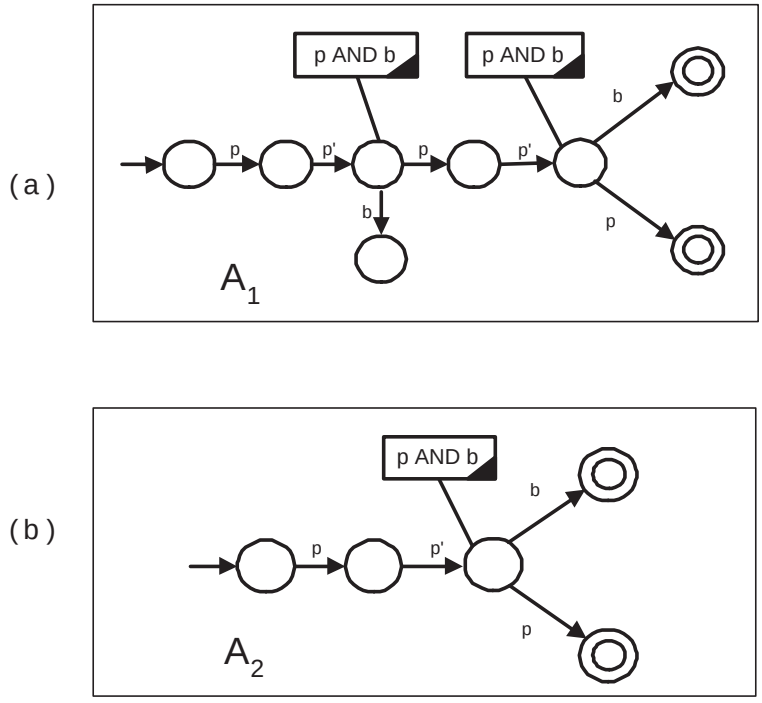


Figure 4.10: False Matches due to Annotations

Figure 4.10 shows an example where two aFSAs A_1 and A_2 are to be matched. Clearly the two aFSAs do not match because they do not share at least one common execution path leading

Table 4.7: Annotations for aFSAs A_1 and A_2 using 2-grams

aFSA	2-gram (ℓ)	2-gram id	annotation (g)
A_1	\$\$	ℓ_1	{p }
A_1	\$p	ℓ_2	{p' }
A_1	pp'	ℓ_3	{b, p }
A_1	p'p	ℓ_4	{p' }
A_1	pp'	ℓ_5	{b, p }
A_2	\$\$	ℓ_6	{p }
A_2	\$p	ℓ_7	{p' }

to a final state, even before evaluation of logical expressions is considered.

However aFSA A_1 contains two states having the same context and the same annotation. The context of these two states is p' for a look-back of one and the annotation is $p AND b$. Since the two states share the same context and annotation, they cannot be differentiated when represented in the repository as shown in the annotations table (Table 4.7). Thus if another aFSA like A_2 as shown in Figure 4.10 (b) is matched, a false match will occur. In this example (Figure 4.10) A_1 and A_2 will cause a false match because the 2-gram pp' (assuming we are using 2-grams) which is associated to the annotation $\{b, p\}$ is not differentiated in A_1 as it occurs in two different states. Notice that increasing the look-back will not resolve this false match because the two contexts will always remain equivalent. Thus false matches may occur due to annotations whenever there exists at least two equivalent n-grams sharing the same annotations in an aFSA description. However such aFSAs are very unlikely in reality, thus does not limit our approach for the problem domain.

4.3.7 False Miss Analysis

While annotations can give rise to false matches as described in Section 4.3.6, false misses due to annotations are not possible since no abstraction is applied to the annotations, hence no information is lost. Moreover, the abstraction ϕ_n does not affect annotations in aFSAs. The bit vectors which are used to represent annotations do not represent an abstraction, but rather a more convenient way to represent logical expressions in the repository such that they can easily be processed. The bit operators which are used to process queries support exactly the same set operations such as set intersection (via bitwise *AND*) and set union (via bitwise *OR*). We analyze the message sequence representation mechanism for potential false misses, since an abstraction ϕ_n is applied to the original aFSA, and information is lost due to the application of ϕ_n.

The abstraction $\tau(\Psi(\Phi_n(L(A))))$ (represented by $\phi_n(L(A))$) ensures that no false misses will occur during search operations. We provide a proof that applying the abstraction does not result in false misses during searching.

Lemma 4.3.1
The n-gram set abstraction of an aFSA language guarantees no false misses during search

operations

Proof:
Let A_1 and A_2 be aFSAs to be matched, where $L(A_1)$ and $L(A_2)$ are languages of A_1 and A_2 respectively. Let $\tau(\Psi(\Phi_n(L(A_1))))$ and $\tau(\Psi(\Phi_n(L(A_2))))$ be abstractions of $L(A_1)$ and $L(A_2)$ respectively. The goal of the proof is to show that if the original aFSAs A_1 and A_2 match by having a common message sequence covering all mandatory messages, their abstractions will also match by having a common abstracted message sequence covering mandatory messages. Formally, we want to show that

$$L(A_1) \cap L(A_2) \neq \emptyset \longrightarrow \tau(\Psi(\Phi_n(L(A_1)))) \cap \tau(\Psi(\Phi_n(L(A_2)))) \neq \emptyset \tag{4.5}$$

From $L(A_1) \cap L(A_2) \neq \emptyset$ follows that there exists $\omega \in L(A_1)$ and $\omega' \in L(A_2)$, with $\omega = \omega'$. The function Φ_n is bijective, so $\Phi_n(\omega) = \Phi_n(\omega')$. If $\Phi_n(\omega) = \Phi_n(\omega')$, the same duplicates are contained, and hence, the same duplicates are removed by Ψ. Therefore, after removing duplicates, we will have $\Psi(\Phi_n(\omega)) = \Psi(\Phi_n(\omega'))$. The function τ is a set abstraction that removes order in the n-gram lists of $\Psi(\Phi_n(L(A_1)))$ and $\Psi(\Phi_n(L(A_1)))$, and uses the set union operator to add n-grams from $\Psi(\Phi_n(L(A_1)))$ and $\Psi(\Phi_n(L(A_1)))$, to their respective sets. Since $\Psi(\Phi_n(\omega)) = \Psi(\Phi_n(\omega'))$, $\Psi(\Phi_n(L(A_1)))$ and $\Psi(\Phi_n(L(A_1)))$, contain the same n-grams, hence $\tau(\Psi(\Phi_n(\omega))) = \tau(\Psi(\Phi_n(\omega')))$ which is contained in $\tau(\Psi(\Phi_n(L(A_1)))) \cap \tau(\Psi(\Phi_n(L(A_2))))$, proving the lemma.

We have illustrated that using a series of abstractions Φ_n, Ψ and τ, on an aFSA represented by its language, the resulting language is finite. We have also shown that using the series of abstractions, no false misses are introduced when matching aFSAs. In addition, we have illustrated that false matches can be kept minimal by using look-back. Thus the series of abstractions Φ_n, Ψ and τ represents the abstraction we are looking for as given in Equation (4.1). In summary for an aFSA A:

$$\phi_n(L(A)) \equiv \tau(\Psi(\Phi_n(L(A)))) \tag{4.6}$$

For the rest of the dissertation, we will use ϕ_n, in place of Φ_n, Ψ and τ as the abstraction function without elaborating further. From an operational point of view, before we insert aFSAs into the database, we will transform them using ϕ_n, to get a finite language.

4.4 Indexing annotated Finite State Automata

The indexing mechanism for aFSAs is based on an index representing message sequences of a collection of aFSAs and another index for representing annotations for the same aFSA collection. Each of the two indexes can independently be queried, the intermediary results of which

can be combined to a final result set. The message sequence index is a set of pairs (ℓ', D') where ℓ' is an n-gram and D' is a set of all aFSAs with n-gram ℓ' in the repository. The annotation index is a set of tuples (D', ℓ', g) where D' is a set of all aFSAs with n-gram ℓ' and g is a bit vector representation of an annotation. This section describes the indexing mechanism for aFSAs in detail and how to evaluate queries based on message sequence and annotation indexes.

4.4.1 Message Sequence Index

The resulting n-grams constituting the finite language computed using ϕ_n are added to the database, which can be indexed using standard indexing techniques such as a B^+-trees for efficient searching or matchmaking. The index is built by calculating the finite set of n-grams and relating them to aFSAs. The index key used for sorting is the n-gram string.

Formal Definitions

The index based on message sequences is formally described in the next definition:

Definition 4.4.1 (Message Sequence Index)
If the database is a collection D of aFSAs with $D := \{A_1, \ldots, A_N\}$. *The message sequence index* I_M *is:*

$$I_M := \{(\ell', \bigcup_{A_i \in D \wedge \ell' \in \phi_n(L(A_i))} \{A_i\})\} \tag{4.7}$$

The *message sequence index* is constructed by mapping n-grams to aFSAs. Queries can be performed on the basis of n-grams, without taking into account the annotations, for the time being. The query is performed by transforming the query aFSA using the same abstraction ϕ_n on the query aFSA. Matching aFSAs are found by checking for equivalent n-grams within the index by a string comparison in the database. The query evaluation is formally defined as:

Definition 4.4.2 (Query Evaluation)
Let $D := \{A_1, \cdots, A_N\}$ *be the set of aFSAs in the collection D and A the query aFSA. The query result R contains aFSAs* A_i *having at least one n-gram in common*

$$R(A) := \bigcup_{p \in \phi_n(L(A))} \left\{ A_i \mid p \in \phi_n(L(A_i)) \right\} \tag{4.8}$$

with

$$\Theta_n(p, A_i) := \begin{cases} \{A_i\} & if\ p \in \phi_n(L(A_i)) \\ \emptyset & otherwise \end{cases}$$

As we have shown in Lemma 4.3.1 above, no false misses are introduced by the abstraction during query operations.

Operationalization

In this section we describe how the indexing mechanism is used for evaluating queries. The *message sequence index* - I_M which is given in Equation 4.7 can be searched based on the n-gram attribute as described formally below:

Definition 4.4.3 (Searching Message Sequence Indexed aFSAs)
Let $D := \{A_1 \cdots A_N\}$ be a collection of aFSAs indexed using I_M, the message sequence index and A be a query aFSA to the collection D. $R_1(\ell') \subseteq D$ is the set of aFSAs matching A with $\ell' \in \phi_n(L(A))$ where $R_1(\ell')$ is denoted in relational algebra as:

$$R_1(\ell') := \Pi_{\{\mathbf{aFSA}\}}\left(\sigma_{(\ell'=\ell)}(t_1)\right) \tag{4.9}$$

where t_1 is a relation comprising aFSAs, and their associated n-grams, aFSA is an attribute representing an aFSA in t_1 and ℓ is an attribute representing an n-gram in table t_1.

The symbols Π and σ are the usual relational operators of projection and selection respectively. Equation 4.9 states that given a relation t_1 of the form $t_1(\mathbf{aFSA}, \ell)$, constructed based on the index I_M, where the attributes **aFSA** and ℓ represent the aFSA and n-gram respectively, with ℓ computed, for each aFSA, using ϕ_n, matching aFSAs are found for an n-gram ℓ' of the query aFSA, and checking for equivalent n-grams from the database collection. From the table resulting from applying the *select* operator, the *project* operator is used to get the set of matching aFSAs.

Complexity

We analyze the message sequence indexing approach by examining the computational and storage complexities for setting-up or constructing the indexes and performing search operations. The calculations for the n-gram indexes are based on the number of production rules contained in the grammar and the number of n-grams contained in the finite abstracted language. The production rules are generated by traversing aFSAs in the collection. The number of traversals depends on the structure of the aFSA and the look-back n. For example the number of productions for a non-cyclic aFSA is obtained by iterating over the aFSA in time linear to the number of transitions in the aFSA. However, iterating over a cyclic aFSA is much more complex because the cycles must be traversed several times, depending on the look-back to compute all production rules. The complexity is even higher for aFSAs with nested cycles because the number of production rules rapidly increases due to the nesting of cycles as every possible combination of paths must be traversed.

In this dissertation, we differentiate between three types of aFSAs depending on their structure. They are (i) aFSAs with no cycles (ii) aFSAs with simple cycles and (iii) aFSAs with complex cycles. An aFSA with no cycles has a finite language. Such aFSAs are easy to process and to index because their languages can be easily enumerated and indexed using standard techniques for string equivalence. The cycles of an aFSA graph are either simple or complex [Weinblatt, 1972]. We formally describe cyclic aFSAs below.

Definition 4.4.4 (Cyclic aFSAs)
Let $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ *be an aFSA. A is cyclic if there exists at least one cycle* $c =< t_1, \cdots, t_n >\in \Delta^n$ *with* $t_i = (q_i, a_i, q_{i+1})$ *for all* $1 \leq i \leq n-1$ *and* $t_n = (q_n, a_n, q_1)$*. Let* $C(A)$ *be the set of all cycles contained in automaton A.*

Definition 4.4.4 above describes an ordered set of transitions such that the first and last transitions in the list share a common source and target state respectively. In the next definition we describe simple cycles.

Definition 4.4.5 (Simple Cycles)
Let $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ *be an aFSA. A contains only simple cycles if all its cycles have disjoint states. Formally, A contains only simple cycles if for all* $c_\ell, c_r \in C(A)$ *with* $c_\ell =< t_{\ell,1}, \cdots, t_{\ell,n_\ell} >\in \Delta^{n_\ell}$ *and*
$c_r =< t_{r,1}, \cdots, t_{r,n_r} >\in \Delta^{n_r}$ *the following holds*

$$\nexists \quad i \in \{1, \cdots, n_\ell\}, j \in \{1, \cdots, n_r\}. q_{\ell,i} = q_{r,j} \wedge \\ t_{\ell,i} = (q_{\ell,i}, a_{\ell,i}, q_{\ell,i+1}) \wedge t_{r,j} = (q_{r,j}, a_{r,j}, q_{r,j+1})$$

Every cycle for which there exists another cycle, which share the same state is called a *complex cycle*. The complexity analysis in this dissertation is based on aFSAs with simple cycles. This is due to the fact that business processes, which are motivating this research, are based on aFSAs with no cycles and aFSAs with simple cycles.

We have already pointed out that aFSAs with no cycles are easy to index because their message sequences can be enumerated. As a result they exhibit the least complexity in terms of the number of message sequences. The languages of aFSAs with simple or complex cycles are both infinite, however, the languages of aFSAs with complex cycles usually contain a larger number of words up to a certain maximum length than languages associated to aFSAs with simple cycles due to the combination of the different complex cycles.

We will use acyclic aFSAs for computing the best case complexity. In addition to analysing the computational complexity of aFSAs without cycles, we will also analyse the computational complexity of aFSAs with simple cycles. We will also examine aFSAs with complex cycles, but the focus of this work will be on aFSAs with simple cycles. The calculation of the number of production rules *R(n)* and the number of n-grams *W(n)* for acyclic and simple cyclic aFSAs is

discussed first before the results are used for the index construction and index search complexities.

The best case complexity for the number of n-grams to be stored is when an aFSA is non-cyclic. The number of productions $R_{min}(n)$, generated for a non-cyclic aFSA is given by

$$R_{min}(n) = 2 + |\Delta| + |F| \tag{4.10}$$

where $|\Delta|$ is the number of transitions and $|F|$ is the number of final states. The rational for the above formula is as follows: every aFSA transition (q_1, a, q_2) results in a new production $a_1 \cdots a_{n-1}[q_1] \xrightarrow{a} a_2 \cdots a_{n-1}a[q_2]$ where n is the look-back and every aFSA transition leading to a final state results in an extra production $a_1 \cdots a_{n-1}[q_2] \xrightarrow{\#} a_2 \cdots a_{n-1}\#$. In addition, there are two start productions of the form $\varepsilon[s] \xrightarrow{\varepsilon} \$^n[s']$ and $\$^n[s'] \xrightarrow{\$} \$^n[q_0]$ where q_0 is a start state.

The minimum number of n-grams $W_{min}(n)$ cannot exceed the number of messages $|\Sigma|$ plus two in the best case, which is when the aFSA is acyclic. Hence, the number of n-grams generated $W_{min}(n)$ for the best case is:

$$W_{min}(n) \quad \leq \quad 2 + |\Sigma| \tag{4.11}$$

The rational for Equation 4.11 is as follows: each n-gram is constructed by looking-back a predefined number of steps n, and prefixing the message with n previous messages. This means that n-grams for the aFSA are constructed by prefixing each message with n previous messages, with duplicate n-grams being removed. Thus the resulting number of n-grams created this way cannot exceed the number of messages in the aFSA. In addition, there are two special n-grams containing the $ and # symbols, and associated with the start and final states respectively. The extra special n-grams explain the two in Equation 4.11 above.

The next definition presents a derivation for the number of productions in an aFSA simple cycle. Let A be an aFSA where the most complex simple cycle has $|\Delta'|$ transitions. For a look-back n, the maximum number of productions for this simple cycle is computed recursively as follows:

$$
\begin{aligned}
R(0) &\leq 2 + |\Delta'| + |F| \\
R(1) &\leq R(0) * |\Delta'| \\
R(2) &\leq R(1) * |\Delta'| \\
&\leq (R(0) * |\Delta'|) * |\Delta'| \\
&\leq R(0) * |\Delta'|^2 \\
&\vdots \\
R(n) &\leq R(0) * |\Delta'|^n \\
&\leq (2 + |\Delta'| + |F|) * |\Delta'|^n \\
&\leq (2 * |\Delta'|^n + |\Delta'|^{n+1} + |\Delta'|^n * |F|)
\end{aligned}
$$

$R(n)$ can be expressed in terms of the complexity for computing productions in aFSA simple cycles, which is given in Big-O notation as:

$$R(n) = O\left(|\Delta'|^{n+1}\right) \tag{4.12}$$

The rational for the derivation of Equation 4.12 is as follows: the number of rules when look-back is zero cannot exceed the sum of the number of aFSA transitions, final states plus two as already described for Equation 4.10 above. Thus the maximum number of productions $R(0)$ for a look-back of zero is $R(0) = 2 + |\Delta'| + |F|$. A new grammar for a given look-back n, is computed from the previous grammar (look-back amount $n-1$). This requires iterating over the grammar rules for the previous look-back $n-1$. In each iteration, every grammar rule can generate at most $|\Delta'|$ new rules, hence the total number of new rules is the product of total number of previous rules and the number of transitions (assuming that each transition results in the introduction of a new rule). The termination condition is the call to $R(0)$, which is computed non-recursively. Thus the complexity for computing the number of rules in aFSA simple cycles is $O(|\Delta'|^{n+1})$, being exponential on the look-back n. It is worth pointing out that the typical value of $|\Delta'|$, which is the number transitions in a cycle, where the longest cycle in an aFSA is considered, is rather small. Typical maximum cycle lengths for RosettaNet data used in the experiments ranged between 2 and 10. The look-back is also a small number ranging between 0 and 5. Thus in practice, the exponential result, $O\left(|\Delta'|^{n+1}\right)$, does not affect the computation effort in a significant way.

The complexity for the number of n-grams is directly computed from the set of production rules. If $\overline{|\Delta'|}$ is the average number of transitions in the most complex cycle of each aFSA in the collection, $\overline{|\Delta'|}$ is given by:

$$\overline{|\Delta'|} = \frac{\sum_{i=1}^{N} |\Delta'_i|}{N} \tag{4.13}$$

where N is the number of aFSAs in the collection. The number of n-grams that can be generated from the grammar rules of an aFSA with simple cycles can be approximated by $O(\overline{|\Delta'|}^{n+1})$. This result is due to the fact that each n-gram is computed directly from a production rule, and the highest complexity is influenced by cycles, in particular long cycles.

The effort needed to construct the index for an aFSA with simple cycles is a sum of the effort to compute the grammar, the finite set of n-grams and the effort to insert into an index structure such as B$^+$-tree. The best case effort (the effort for aFSAs with no cycles) needed to construct the index is:

$$O(N * \overline{|\Delta|} + log(N * \overline{|\Delta|})) \tag{4.14}$$

where $\overline{|\Delta|}$ is the average number of transitions in an aFSA and N is the number of aFSAs to be inserted into the collection. $O(log(N * \overline{|\Delta|}))$ is the effort needed to insert into a B$^+$-tree index structure [Gray and Reuter, 1993]. The complexity to construct an index for aFSAs with simple cycles can be approximated by:

$$O\Big(N * \overline{|\Delta'|}^{n+1} + log(N * \overline{|\Delta'|}^{n+1})\Big) \tag{4.15}$$

where n is the look-back, $\overline{|\Delta'|}$ is the average maximum cycle length for a collection of size N as given in Equation 4.13. The complexity $O\Big(N * \overline{|\Delta'|}^{n+1} + log(N * \overline{|\Delta'|}^{n+1})\Big)$ is obtained by summing-up the complexity for constructing the index and that for inserting into a B$^+$-tree. The complexity for constructing the index for aFSAs with simple cycles is obtained by multiplying the number of aFSAs in the collection N by the maximum number of n-grams generated for the most complex cycle of each aFSA to obtain $O\Big(N * \overline{|\Delta'|}^{n+1}\Big)$. The complexity for inserting into a B$^+$-tree is obtained by taking the log of $N * \overline{|\Delta'|}^{n+1}$, the estimated total number of n-grams to be inserted.

We now discuss the message sequence index search complexity for aFSAs with simple cycles. Index-based search is done in three phases: (i) transforming the query aFSA into a finite set of words (ii) searching on a standard index structure such as B$^+$-tree (iii) merging intermediate results. The complexity for transforming an aFSA as stated above, is:

$$O(|\Delta_q|) \tag{4.16}$$

for query aFSAs with no cycles and

$$O(|\Delta'_q|^{n+1}) \tag{4.17}$$

for the query aFSAs with simple cycles; where $|\Delta_q|$ is the number of transitions in the query aFSA and $|\Delta'_q|$ is the number of transitions in the most complex cycle of the query aFSA.

The best case complexity for searching on a B$^+$-tree is when aFSAs in the collection are

non-cyclic. The search complexity for aFSAs with no cycles is given by:

$$O(|\Delta_q| * log(N * \overline{|\Delta|})) \tag{4.18}$$

In Equation 4.18 above, the number of n-grams $|\Delta_q|$ from the query aFSA is multiplied by $log(N * \overline{|\Delta|})$, the complexity for searching, because each search operation takes $log(N * \overline{|\Delta|})$ computational effort. The search complexity for aFSAs with simple cycles is:

$$O\Big(|\Delta'_q|^{n+1} * log(N * \overline{|\Delta'|}^{n+1})\Big) \tag{4.19}$$

where N is the total number of aFSAs in the collection, $\overline{|\Delta'|}$ is the average number of transitions for the longest cycle of each aFSA in the collection. As pointed out already, the values of $|\Delta'|$ and n are typically small (n ranges between 0 and 5, $|\Delta'_q|$ ranges between 2 and 10). Thus the exponential complexity does not affect the computation in a significant way.

We have so far presented the complexities for constructing, transforming the query aFSA, searching and merging results for the message sequence index. We now combine the complexity calculations to determine the overall complexity for the message sequence index with simple cycles. The total search complexity for aFSAs with no cycles is computed by summing-up the construction, transformation, searching and merging complexities. The overall search complexity for the message sequence index for aFSAs with no cycles is:

$$O\Big(|\Delta_q| + |\Delta_q| * log(N * \overline{|\Delta|}) + N' * logN')\Big) \tag{4.20}$$

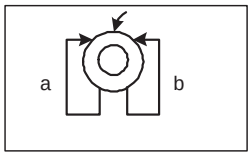
The overall search complexity for aFSAs with simple cycles is similarly obtained, being:

$$O\Big(|\Delta'_q|^{n+1} + |\Delta'_q|^{n+1} * log(N * \overline{|\Delta'|}^{n+1}) + N' * logN')\Big) \tag{4.21}$$

The search complexity for aFSAs with simple cycles is exponential on the look-back. As we pointed out before, the typical look-back is rather small, ranging from 0 to 5, and the maximum cycle length $\overline{|\Delta'|}$ is also small with values ranging from 2 to 10. The exponential complexity has no significant effect on the computation time.

We have so far considered message sequence indexing for aFSAs with simple cycles. The number of productions for aFSAs with complex cycles quickly explodes. This is because in aFSAs with complex cycles, cycles are nested, resulting in a combinatorial explosion of the potential number of productions to be used to generate new productions. In the worst case, every transition in the aFSA with complex cycles constitutes a nested cycle. However the number of n-grams to be stored is reduced because for each aFSA, n-gram sets are stored in the database, rather than n-gram sequences. The set abstraction minimizes the number of n-grams to be stored, thus reducing the overall storage complexity, which also reduces the number of comparisons to be made during search operations.

Figure 4.11 shows an example aFSA with complex cycles. In Figure 4.11 (a), $|\Delta|$ repre-



(a)

$\|\Delta\|$	n	$\|P\|$	W(n)
2	0	7	5
2	1	15	53
2	2	31	2 555
2	3	63	3 377 733
2	4	127	Out of memory
2	5	255	Out of memory

(b)

Figure 4.11: aFSA with Complex Cycles

sents the number of transitions, n, the look-back, $|P|$, the number of productions and $W(n)$ the number of words represented by n-gram sequences. The figure shows that for an aFSA with two complex cycles, the number of productions grows by more than a factor of two, when the look-back is increased in steps of one. In addition, the number of words grows exponentially as the look-back n increases. When $n = 4$ and beyond an *out of memory error* was returned on a Pentium 4 processor Dell machine with 1000 MB RAM. Not all of this memory was available for processing the sequences as only a fraction of the RAM was allocated to the Java Virtual Machine (JVM). The out of memory problem can be resolved by having additional computational resources especially memory, and tuning the virtual machine to process larger data sets at a time. Another option to resolve the out of memory problem is to implement the algorithm for generating sequences non-recursively, using iteration and relying on external data structures for storing intermediate results. The two approaches do not represent an optimal solution for handling complex cycles. A more convenient approach is needed to handle aFSAs with complex cycles for applications where complex cycles are critical.

The approach presented in this dissertation is applicable to aFSAs with simple cycles in general and complex cycles in case of look-back $n = 0$ or $n = 1$. This suffices for the problem motivated in this research as business processes can have no cycles or can have simple cycles but complex cycles are rare.

4.4.2 Indexing Logical Annotations

Logical annotations are represented using fixed-length bit vectors and evaluated based on bit vector operations for intersection, union and complement. In the following, the formal definition for the annotation index and the algorithm for searching are presented.

Formal Definition

The formal definition of the annotation index is based on definitions for representing annotations as presented in Section 4.3.5. We begin by giving a formal definition for an annotation index.

Definition 4.4.6 (Annotation Index)
Let the database be a collection D of completely annotated aFSAs with $D := \{A_1, \ldots, A_N\}$ with $A_i = (Q_i, \Sigma_i, \Delta_i, q_{0,i}, F_i, QA_i, role_i)$. The annotation index I_A is:

$$I_A := \bigcup_{A_i \in D} \bigcup_{q \in Q_i} \bigcup_{\ell' \in \theta(q)} \bigcup_{i=1}^{m} \left\{ (A_i, \ell', g_i) \right\} \tag{4.22}$$

where $(q,e) \in QA_i$ and $e := g_1 \wedge \cdots \wedge g_m$ with d_i being the bit vector representations of the disjuncts of the expression e in Disjunctive Normal Form (DNF).

The annotation index is a set of triples of aFSAs, n-grams and annotations, where each n-gram is associated to an annotation. The outer union is needed to iterate over aFSAs in the collection. For each aFSA, we iterate over states (second union operation), and for each state, associated annotations are collected (third union operation), and from each annotation, we iterate over disjuncts $e := g_1 \wedge \cdots \wedge g_m$. Subsequently, the set $\left\{ (\{A_i\}, \ell', g) \right\}$ associating aFSAs, n-grams and annotations, is constructed. Note that we have made the assumption that the logical expression e is expressed in disjunctive normal form as previously stated elsewhere. The annotations are represented as bit vectors, and bit operations are used for query evaluations as will be illustrated later in this chapter.

Operationalization

This section describes the use of the annotation index I_A, described in Definition 4.4.6 for the evaluation of annotations during search operations. The query used in the algorithm is based on the n-gram ℓ' representing the current state and the bit vector representation of a disjunct g' of the annotation of the current state. The query can be denoted in relational algebra as follows:

Definition 4.4.7 (Searching Annotation Indexed aFSAs)
Let $D := \{A_1 \cdots A_N\}$ be a collection of aFSAs indexed using I_A, an annotation index and $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ is a query aFSA to D. $R_2(\ell', g') \subseteq D$, where $\ell' \in \theta(q)$, and $g' \in \rho(q)$, with $q \in Q$, computes the set of aFSAs that contain aFSAs with equivalent n-grams to the query

aFSA, where the mandatory message requirements of the query aFSA are not fulfilled. $R_2(\ell', g')$ *is expressed in relational algebra as:*

$$R_2(\ell', g') := \Pi_{\{\mathbf{aFSA}\}}\big(\sigma_{\substack{(\ell = \ell') \\ (g \cap g' \neq \emptyset) \\ (g \cap \bar{g'} \neq \emptyset)}} \; \wedge \; \wedge \; (t_2)\big) \tag{4.23}$$

where $\bar{g'}$ *is the negation of* g' *and* $t_2 = (aFSA, \ell, g)$*, with* t_2 *being a relation based on the* I_A *index.*

The *selection* operator has three predicates that are all joined by a logical **AND** operator. The predicate $\ell = \ell'$ selects aFSAs from the collection with the same n-grams as the query aFSA. The predicate $g' \cap g \neq \emptyset$ selects aFSAs with an annotation that share the same variable term with the corresponding query annotation. An example of such an annotation is a **AND** c and a **AND** b, mapping to the sets $\{a, c\}$ and $\{a, b\}$ respectively, where one of the annotations belongs to an aFSA from the collection and the other to the query aFSA. Based on the sets $\{a, c\}$ and $\{a, b\}$, the variable a is shared by both aFSAs, so the predicate evaluates to a non-empty set. The predicate $d \cap \bar{d'} \neq \emptyset$ is needed to ensure that only aFSAs whose annotations completely match, are added to the result set. This is achieved by checking that for conjunction, a matching annotation will contain the same conjuncts. For example, a **AND** c and a **AND** b must not match because though they contain a common variable a, the other conjuncts are different, thus are not considered a solution of the matchmaking. In other words, a service finder who is insisting on the support for messages a and b (a **AND** b) is not satisfied to receive a and c (a **AND** c), because even though a is supported, c is not. The effect of all the three predicates is to eliminate aFSAs that have matching n-grams, but with corresponding annotations that do not match.

Complexity

In this sub-section we describe the computational complexity for setting-up the annotation index and performing search operations on the index. We will assume a database collection with N aFSAs. The effort needed to construct the annotation index is the sum of the effort to transform logical expressions into appropriate groups of message sets, that are then represented as bit vectors, and the effort to insert into a B^+-tree index structure. The complexity for transforming logical expressions is derived from the formula given by Equation 4.4.

Let $A = (Q, \Sigma, \Delta, q_0, F, QA, role)$ be an aFSA where for each $(q, e) \in QA$, where e is a logical expression in disjunctive normal form. The effort to transform the logical expressions is

$$O\Big(|Q| * \overline{|g|}\Big) \tag{4.24}$$

where $|Q|$ is the number of states and $\overline{|g|}$ is the average number of disjuncts in each logical expression e. The rational behind Equation 4.24 is that transforming disjuncts to bit vectors

requires time linear on the number of disjuncts. Thus to get the overall complexity, we multiply the average number of disjuncts by the number of states in an aFSA resulting in a computational complexity of $O\left(|Q| * \overline{|g|}\right)$. Thus transformation of logical expressions to bit vector format can be done in time linear to the number of states. The computation effort to insert into a database is

$$O\left(log(N * \overline{|Q|} * \overline{|g|}\right) \tag{4.25}$$

where $\overline{|Q|}$ is the average number of states in each aFSA, $\overline{|g|}$ is the average number of disjuncts in each logical expression and N is the number of aFSAs in the collection. The product $N * \overline{|Q|} * \overline{|g|}$ gives the approximate number of annotations to be added to the collection. Inserting into a B^+-tree requires logarithmic time, explaining Equation 4.25. The total computation complexity for setting-up the annotation index is a summation of the complexity for transformation of annotations and insertion into the database which is

$$O\left((N * \overline{|Q|} * \overline{|g|}) + log(N * \overline{|Q|} * \overline{|g|})\right) \tag{4.26}$$

The complexity for searching is computed by summing-up the complexity to evaluate expressions of the query aFSA and the complexity to perform search operations. The complexity for evaluating expressions is linear on the number of elements in the expression operation tree. In the worst case, the number of leaf nodes of the operation tree is $2^{|\Sigma|}$, which is the order of the power set of the message set Σ. The number of leaf nodes $2^{|\Sigma|}$ can be expressed as 2^h with h being the tree height. The total number of internal nodes for a tree with degree k is $\frac{k^h-1}{k-1}$. Thus the total number of internal nodes is $2^{|\Sigma|} - 1$ after substituting h with Σ for $k = 2$. Thus the overall total number of nodes (including leaf and internal nodes) is $2^{|\Sigma|} + 2^{|\Sigma|} - 1$ which reduces to $2^{|\Sigma|+1} - 1$. Since expressions are assigned to states as already explained before, the overall worst case complexity of evaluating expressions can be stated in Big-O notation as

$$O\left(\overline{|Q|} * (2^{|\Sigma|+1} - 1)\right) \tag{4.27}$$

where $|\Sigma|$ is the number of messages in the collection. Real-life business processes have a limited number of annotations (with typical values around 5 annotations in a state), thus this complexity is not likely to have a significant influence on the overall computation time during search operations.

4.4.3 Index Search

We have so far specified how each of the two indexes described in this dissertation is used during query evaluation. In this section we now describe an algorithm for searching based on the two indexes to support matchmaking queries. The search algorithm is based on a recursive traversal of the query aFSA, evaluating the n-grams and annotations at each state, and using these to compute the intermediate results using the message sequence and annotation indexes. The intermediate results in each state are intersected with subsequent results, until a final state

is reached.

Algorithm

Before presenting the algorithm, we first describe terms and functions that are used within the algorithms.

- **source**(**t**) - aFSA source state of a transition t,
- **target**(**t**) - aFSA target state of a transition t,
- **label**(**t**) - aFSA message label of a transition t,
- **Var**(**expr**) - the set of variables in an expression **expr**, at a given aFSA state.

Query aFSAs are pre-processed before applying the algorithms described in this section such that all final states of query aFSAs have no out-going transitions. The goal of pre-processing query aFSAs is to ensure that during the traversal of a query aFSA graph, all transitions will be reached. If final states have outgoing transitions, some transitions will not be reached as the recursion stops as soon as a final state is found, meaning all transitions coming after the final state will be ignored (see *Algorithm A*). The algorithm for pre-processing query aFSAs to avoid this problem is described in detail in Appendix A.

The algorithm for searching using the indexing mechanism is presented in **Algorithm A**. The input parameter q is a query aFSA state, with initial value q_o being the start state. *Val* is an array, containing intermediate results for a particular state. *Val* has initial value $\emptyset$. *Vis* is a container to which visited states are added. The initial value of *Vis* is $\emptyset$. The algorithm first checks if the current state is a final state. If the current state is a final state, a function **dbQuery** to evaluate the partial query for the current state is called.

The function *dbQuery* accepts an aFSA state as input. It first initializes a collection R to $\emptyset$ (line 2 of dbQuery), and iterates over the n-grams in that state. Note that the input aFSA is transformed using algorithms described in Section 4.3.4 to construct an n-gram-based language abstraction of the input query aFSA. States can be mapped to n-grams using a hashtable to speed-up look-up operations. Inside the **for** $-$ **loop**, table t_1 is queried to return the set of aFSAs with matching aFSAs (line 4 of dbQuery). By table t_1, we refer to the table based on the message sequence index I_M described in Definition 4.4.1. In line 5 of dbQuery, table t_2 is queried to return the set of aFSAs R_2 that have an equivalent n-gram ℓ' with the query aFSA at state q, but having annotations that do not match thus are unwanted. The argument for line 5 (of dbQuery) was given when Equation 4.23 was introduced. In line 6 of dbQuery all aFSAs R_2 with n-grams matching the aFSA query at state q, but whose annotations are not matching are removed from R_1 and the result is added to R. The results R for the state q are returned to the calling function in line 8.

In **Algorithm A** lines 5 represents entry into a loop over transitions of the query aFSA. If the target state of the current transition is not already visited, lines 7 - 13 are executed. First,

the target state of the current transition is added to the set of visited states Vis in line 7. Next the algorithm checks if the target state of the current transition already has matching aFSAs references. If not, a recursive call is made with the current state set to the target of the current transition (line 10). The Val array, with index set to the target state of the current transition is assigned to the result of the recursive call (line 11). In line 12 the result returned by dbQuery (described in the last paragraph) is intersected with the value of the last recursive call, and the result is mapped to the message label for the current transition (line 13). The meaning of $v := v \cup \{label(t) \longrightarrow ret\}$ in line 13 is that a variable, meaning a transition label, is assigned to

Algorithm A : Matchmaking Query Evaluation

```
1:      evaluate(q, Val, Vis){
2:          if (q ∈ F) return dbQuery(q);
3:          T := {t | source(t) = q};
4:          if (T != null)
5:              for (t ∈ T){
6:                  if (target(t) ∉ Vis) {
7:                      Vis = Vis ∪ {target(t)};
8:                      ret = Val[target(t)];
9:                      if (ret = null) {
10:                         ret = evaluate(target(t), Val, Vis);
11:                         Val[target(t)] := ret;
12:                         ret := ret ∩ dbQuery(q);
13:                         v := v ∪ {label(t) ⟶ ret};
                        }
14:                 } else return dbQuery(target(t));
            }
15:         Res := ∅;
16:         if ((q, expr) ∈ QA)
17:             Res := eval(expr, v);
18:         collectedVars := Var(expr);
19:         if (collectedVars != null)
20:             for (i ∈ collectedVars)
21:                 v := v \ {i};
22:         if (v != ∅)
23:             for (k ∈ v)
24:                 Res := Res ∪ v[k].ret;
25:         return Res;
        }
```

a collection of aFSAs that have been computed for the current transition and target state, and the result is added to the current collection v.

Lines 15 to 24 of **Algorithm A** compute intermediate results of the search query. In particular, for each state of the query aFSA, the logical expression must be evaluated to compute the final results. In line 16, the algorithm checks if an expression is explicitly given for the current state. This is necessary because, as already pointed out in Chapter 3, no logical expressions are explicitly provided for disjunctive choices, because disjunction is the default semantics for choices in automata. If such an expression is explicitly given it is evaluated using a function *eval*

Algorithm B : Evaluating SubQueries

```
1:      dbQuery(q){
2:          R := ∅;
3:          for (each (ℓ', g').ℓ' ∈ φ_n(L(A)) ∧ (g' ∈ ρ(q))) {
4:              R_1 := Π_aFSA ( σ_{ℓ'=ℓ}(t_1) )
5:              R_2 := Π_{aFSA} ( σ_{(ℓ = ℓ') ∧ (g' ∩ g ≠ ∅) ∧ (g' ∩ ḡ ≠ ∅)} (t_2) );
6:              R := R ∪ (R_1 \ R_2);
7:          }
8:          return R;
9:      }


// OR operator:
1:      eval(e_1 ∨ e_2, v){
2:          v1 := eval(e_1, v);
3:          v2 := eval(e_2, v);
4:          return v1 ∪ v2;
        }


// AND operator:
1:      eval(e_1 ∧ e_2, v){
2:          v1 := eval(e_1, v);
3:          v2 := eval(e_2, v);
4:          return v1 ∩ v2;
        }
```

which takes as input an expression and a collection mapping variables to values computed from the function (line 17). The function *eval* evaluates logical expressions using recursive descent parsing by building a parse tree, whose leaves are the set of aFSAs associated to v. The parse tree is evaluated as follows: if an *OR* is encountered, a call to *eval* supporting the *OR* operation is made. Similarly, if an *AND* is encountered, a call to *eval* supporting the *AND* operation is made. The evaluation of *OR* and *AND* operations is shown in *Algorithm B*. A logical expression at an aFSA state may not consider all outgoing transitions labels. So, in line $20-21$ (*Algorithm A*), the algorithm removes, from v all those transition labels which are in the expression. Lines $22-24$ adds the results of those options that had not been considered in the expression, to the final result set.

Search Complexity

The search complexity is influenced by the search complexity for the message sequence index I_M, the search complexity for the annotation index I_A as well as the complexity to merge intermediate results. The best case search complexity is a summation of the best case complexities of the message sequence index and annotation index searches as well as the complexity for merging intermediate results, which is obtained from Equations 4.20 and 4.26 as;

$$O\Big(|\Delta_q| + |\Delta_q| * log(N * \overline{|\Delta|}) + N' * logN' + N * \overline{|Q|} * \overline{|g|} + log(N * \overline{|Q|} * \overline{|g|})\Big) \tag{4.28}$$

where $|\Delta_q|$ is the number of transitions in the query aFSA, N is the size of the data collection, $\overline{|\Delta|}$ is the average number of transitions in the collection, N' is the size of intermediary results to be merged, $\overline{|Q|}$ is the average number of states in the collection and $\overline{|g|}$ is the average number of disjuncts in each logical expression. Equation 4.28 can be rewritten as:

$$O\Big(|\Delta_q| + |\Delta_q| * logN + |\Delta_q| * log\overline{|\Delta|} + N' * logN' + N * \overline{|Q|} * \overline{|g|} + logN + log\overline{|Q|} + log\overline{|g|}\Big) \tag{4.29}$$

From Equation 4.29 and considering that $|\Delta_q|$, $\overline{|g|}$ and $\overline{|Q|}$ are usually small compared to the data collection size N, the best case search complexity can be approximated by $O\Big(logN\Big)$ which represents logarithmic complexity. Even if the other parameters are considered, the search complexity is still logarithmic with respect to the data collection size, making the approach scale as the data set size in increased and other variables are held constant.

The complexity for aFSAs with simple cycles is similarly obtained by a summation of their complexities for message sequence index and annotation index searches (Equations 4.21 and 4.27 respectively), being

$$O\Big(|\Delta'_q|^{n+1} + |\Delta'_q|^{n+1} * log(N * \overline{|\Delta'|}^{n+1}) + N' * logN' + \overline{|Q|} * (2^{|\Sigma|+1} - 1)\Big) \tag{4.30}$$

Equation 4.30 can be reduced to

$$O\Big(|\Delta'_q|^{n+1} + |\Delta'_q|^{n+1} * logN + (n+1) * |\Delta'_q|^{n+1} * log\overline{|\Delta'|} + N' * logN' + \overline{|Q|} * (2^{|\Sigma|+1} - 1)\Big) \quad (4.31)$$

We have already stated that typical values for the look-back n range between 0 and 5, while $\overline{|\Delta'|}$, the average length for the most complex cycle in each aFSA is between 5 and 10. Look-backs between 0 and 5 have been used in experiments with good quality of results in terms of the number of false matches when look-back was 4 with RosettaNet processes. Also, in reality the number of annotations for business processes is rather small (typical values around 5), hence the exponential effect has no significant influence on the overall computation time. Thus the search algorithm has good computational properties for low values of n. In particular search complexity is logarithmically related to the data set size, thus search time increases logarithmically with increasing data set size when all other parameters are kept constant.

The number of false matches occurring during search operations depends on the structure of aFSAs in the data collection. As a worst case scenario, if the search operation returns all aFSAs in the data collection due to false matches, the search time will be comparable to sequential scanning, due to the high number of n-grams to be compared, as the index will not sufficiently prune the search space. The number of false matches is likely to be high for low values of look-back (look-back=0 and look-back=1). The number of false matches is reduced by increasing the look-back as we have pointed out in earlier sections of the dissertation. This will reduce the search space, thus give better performance (and better quality of results) up to a certain look-back. Thus the structure of the data in the collection is an important factor to both the search performance and the quality of search results.

Summary of Search Algorithm

In summary, **Algorithm A** uses depth first search to traverse the query aFSA and returns when a final state has been reached. During the backtracking, in each aFSA state, queries are made to the database to find matching aFSAs based on n-grams on the current state, and those aFSAs that have matching n-grams, but non-matching annotations are removed. The result set at each aFSA state are intersected with those from the previous state along the same path. Meanwhile, a mapping collection such as hash map data structure is used to collect pairs of (**message name variable, result set**) in each state. This is repeated until the start state is reached. Finally logical expressions are evaluated by constructing a parse tree, which is parsed. The hash map collection containing a set of pairs, (**message name variable, result set**) as well as a logical expression are used as input to evaluate logical expressions at each state.

The search algorithm has good computational properties: the best case search complexity is approximately logarithmic to the data collection size while the worst case complexity is also logarithmic to the data set size when all other parameters are held constant.

4.4.4 Search Examples

This section presents examples to illustrate search operations based on the search algorithm presented in Section 4.4.3. We will first present an example using a look-back of zero on a representative database collection. We will extend this example to use a look-back of 1. We will also vary the query aFSAs to illustrate the handling of queries with and without annotations.

Table 4.8: Message Mapping to Bit Vectors of Fixed-Length Eight

message name	bit vector [fixed-length = 8]
a	0000 0001
b	0000 0010
i	0000 0100
$	0000 1000
#	0001 0000

The examples described in this section use the message to bit-vector mapping shown in Table 4.8.

Example 1

Consider the example in Figure 4.12. The database is a collection of aFSAs A_2, A_3 and A_4 where the aFSAs are using 1-grams or a look-back of zero for indexing message sequences. The query aFSA A has exactly the same structure as aFSAs A_2 and one path in common with aFSA A_3 from the database. The query aFSA A, insists that a matching aFSA from the database must support message *i* followed by messages **a** and **b**, meaning **a** and **b** are mandatory at state position 2.

Table 4.9: 1-gram table (t_1) for Figure 4.13

aFSA	1-gram (ℓ)
A_2	$
A_3	$
A_4	$
A_2	i
A_3	i
A_4	i
A_2	a

aFSA	1-gram (ℓ)
A_3	a
A_2	b
A_4	b
A_2	#
A_3	#
A_4	#
A_4	a

Table 4.9 shows the database table t_1 based on the message index I_M for example aFSAs in Figure 4.12. This is a simple table showing which n-grams are found in which aFSAs. Table 4.10 shows the annotations table t_2 which is based on the annotation index I_A, associating aFSAs,

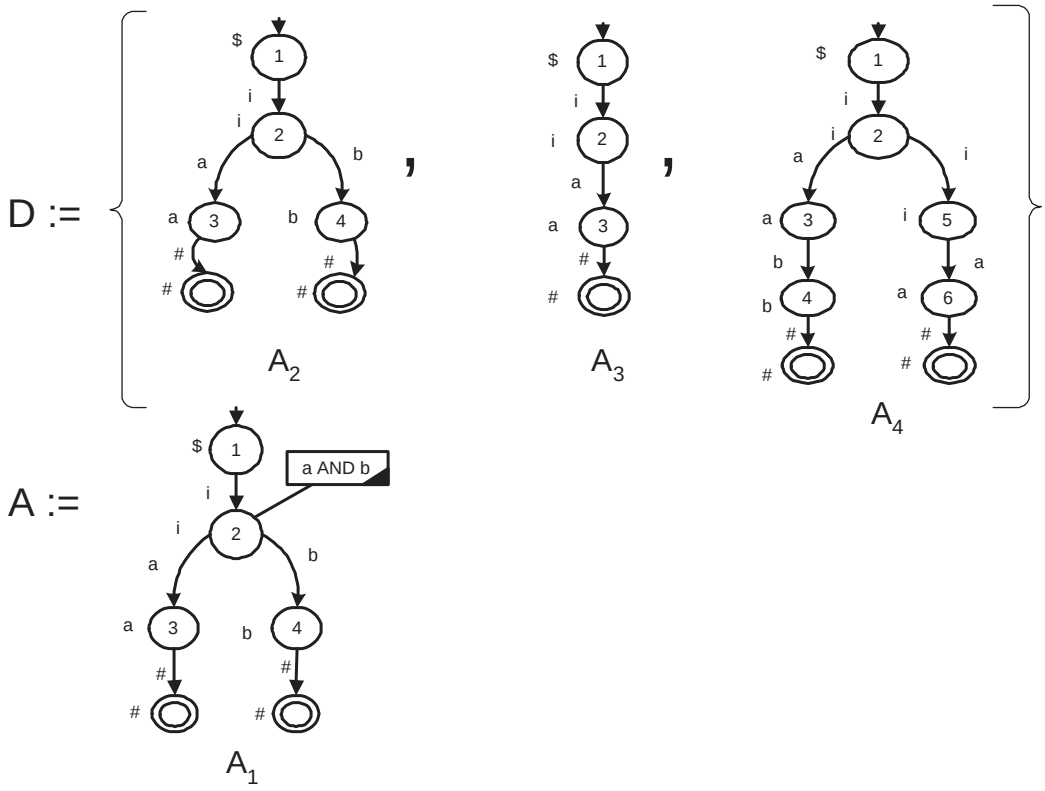


Figure 4.12: Example Showing aFSAs Collection and Query aFSA Based on 1-grams [Look-back of Zero]

n-grams and annotations. Now we can search the two tables using the query aFSA A, using the same abstractions used for the database.

Table 4.11 illustrates the states of variables *Val* and *v* and intermediate result as **Algorithm A** is executed. When in aFSA state 1 which is the start state of the query aFSA, these variables are all empty as shown in the table. The query aFSA is traversed until a final state is reached at which point the algorithm returns. Either path $1 \rightarrow 2 \rightarrow 3$ will be taken first or path $1 \rightarrow 2 \rightarrow 4$. Table 4.11 illustrates the case when path $1 \rightarrow 2 \rightarrow 3$ is taken first. On reaching state 3, the value of *Val* is set to $\{A_2, A_3, A_4\}$ because the n-gram a is found in all three aFSAs from the database collection. Also, a query on the annotations table returns an empty set as there is no annotation for aFSA state 3. So the intermediate result at this state is $\{A_2, A_3, A_4\}$. On backtracking to aFSA state 2 aFSA state 4 is visited and a query for the n-gram b is executed on table t_1.

Only aFSAs A_2 and A_4 have n-gram b; in addition, a query to the annotations table returns an empty set since there are no annotations for aFSA state 4. So *Val* is set to $\{A_2, A_4\}$ and the

Table 4.10: Annotations Table (t_2) for Figure 4.13

aFSA	1-gram (ℓ)	annotation (g)
A_2	$	0000 1000 { $ }
A_3	$	0000 1000 { $ }
A_4	$	0000 1000 { $ }
A_2	i	0000 0001 { a }
A_2	i	0000 0010 { b }
A_3	i	0000 0001 { a }
A_4	i	0000 0001 { a }
A_4	i	0000 0100 { i }
A_4	a	0000 0010 { b }

Table 4.11: Query Evaluation Example 1

State	Val	v	intermediate result	final results
1	$\emptyset$	$\emptyset$	$\emptyset$	
2	$\{A_2,A_3,A_4\}$	$\{ < a \rightarrow \{A_2,A_3,A_4\} >, < b \rightarrow \{A_2,A_4\} > \}$	$\{A_2,A_4\}$	$\{A_2,A_4\}$
3	$\{A_2,A_3,A_4\}$	$\emptyset$	$\{A_2,A_3,A_4\}$	
4	$\{A_2,A_4\}$	$\emptyset$	$\{A_2,A_4\}$	

intermediate results at this point is $\{A_2,A_4\}$. Notice that all the aFSAs in the database collection have the same start n-gram which is a $ in our example. So as an optimization step, the n-gram for the start state is not considered. On state 2, the query for n-gram of *i* is executed; the result of this query comprises all the aFSAs $\{A_2,A_3,A_4\}$ because they all contain the *i* n-gram with no non-matching annotations for this n-gram. Thus *Val* is set to $\{A_2,A_3,A_4\}$ and *v* to $\{ < a \rightarrow \{A_2,A_3,A_4\} >, < b \rightarrow \{A_2,A_4\} > \}$ as shown in Table 4.11.

The next step is to compute final results from intermediate results. This conforms to lines $17-23$ of **Algorithm A**. For this part of the query, the only query aFSA state with a logical expression annotation is aFSA state 2. Notice that in this state, the variable *v* is set to $\{ < a \rightarrow \{A_2,A_3,A_4\} >, < b \rightarrow \{A_2,A_4\} > \}$ which when evaluated results in $\{A_2,A_4\}$ as the matching aFSAs because of the conjunctive annotation.

By inspection, aFSA A_2 is a true match, while A_4 is a false match since A_2 supports the message sequence of A, including the mandatory parts **a AND b**, while A_4 does not. To eliminate the false match A_4, we must increase the look-back.

Example 2

In this example we use a look-back of one to index and search the database. Table 4.12 shows table t_1 for the database collection of Figure 4.13. Figure 4.13 shows the n-grams associated to states.

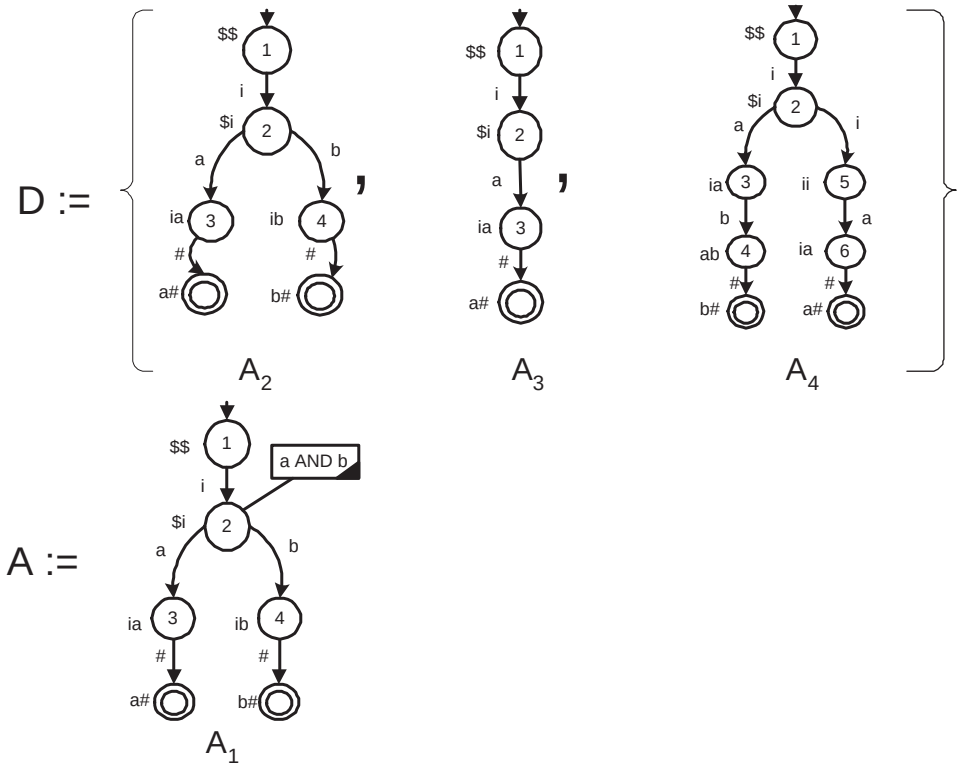


Figure 4.13: Example Showing aFSAs Collection and Query aFSA Based on 2-grams

Table 4.13 represents table t_2 for the database collection of Figure 4.13. The table shows the association of 2-grams of the aFSAs in the collection, with aFSA annotations. Query evaluation proceeds in a similar fashion as described in **Example 1**, except that we are now using a look-back of one, which in principle, should reduce false matches.

Table 4.14 shows states of variables Val, v and intermediate and final results as the query aFSA is recursively traversed. We observe that in Table 4.14 the false match A_4 is eliminated. The result is obtained as follows: by using a look-back of one, a search on table t_1 at position query aFSA state 3 using an n-gram *ib*, we get the set $\{A_2\}$; $\{A_2\}$ is **ANDed** with $\{A_2,A_3,A_4\}$ from state 4, resulting in the set $\{A_2\}$ at state 2. The final result set of $\{A_2\}$ represents a true match.

Table 4.12: 2-gram table (t_1) for Figure 4.13

aFSA	2-gram (ℓ)
A_2	\$\$
A_3	\$\$
A_4	\$\$
A_2	\$i
A_3	\$i
A_4	\$i
A_2	ia
A_3	ia
A_4	ia

aFSA	2-gram (ℓ)
A_2	ib
A_4	ib
A_4	ab
A_4	ii
A_2	$a\#$
A_2	$b\#$
A_3	$a\#$
A_4	$a\#$
A_4	$b\#$

Table 4.13: Annotations Table (t_2) for Figure 4.13

aFSA	2-gram (ℓ)	annotation (g)
A_2	\$\$	0000 0100 $\{i\}$
A_3	\$\$	0000 0100 $\{i\}$
A_4	\$\$	0000 0100 $\{i\}$
A_2	\$i	0000 0001 $\{a\}$
A_2	\$i	0000 0010 $\{b\}$
A_3	\$i	0000 0001 $\{a\}$
A_4	\$i	0000 0001 $\{a\}$
A_4	\$i	0000 0100 $\{i\}$
A_4	ia	0000 0010 $\{b\}$
A_4	ii	0000 0001 $\{a\}$

Example 3

We now illustrate a query where there are annotations in the database. In Figure 4.14, the query aFSA A for the two previous examples is now in the database, while aFSA A_3 is now the query aFSA. So we now have an aFSA with annotations, as part of the database collection. Table 4.15 and Table 4.16 show the n-gram- based table t_1 and annotations table t_2 respectively, for the example database collection in Figure 4.14. Notice that the logical expression **a AND b** in aFSA A_1 is represented as a set $\{a,b\}$ in table t_2 (Table 4.16) and is encoded as bit vector **0000 0011** using the message encoding we introduced in Table 4.8 above.

Table 4.17 shows the state of variables Val, v and intermediate and final results during recursive evaluation of the query, for the case presented in Figure 4.14. All aFSAs in the collection match the query aFSA in this case. However, aFSAs A_2 and A_1 are true matches, while A_4 is a false match. Increasing the look-back will resolve this false match. Thus using the indexing approach described in this dissertation, annotations can also be evaluated within the database.

Table 4.14: Query Evaluation Example 2

State	Val	v	intermediate result	final results
1	$\emptyset$	$\emptyset$	$\emptyset$	
2	$\{A_2, A_3, A_4\}$	$\{ < a \rightarrow \{A_2, A_3, A_4\} >, < b \rightarrow \{A_2\} > \}$	$\{A_2\}$	$\{A_2\}$
3	$\{A_2, A_3, A_4\}$	$\emptyset$	$\{A_2, A_3, A_4\}$	
4	$\{A_2\}$	$\emptyset$	$\{A_2\}$	

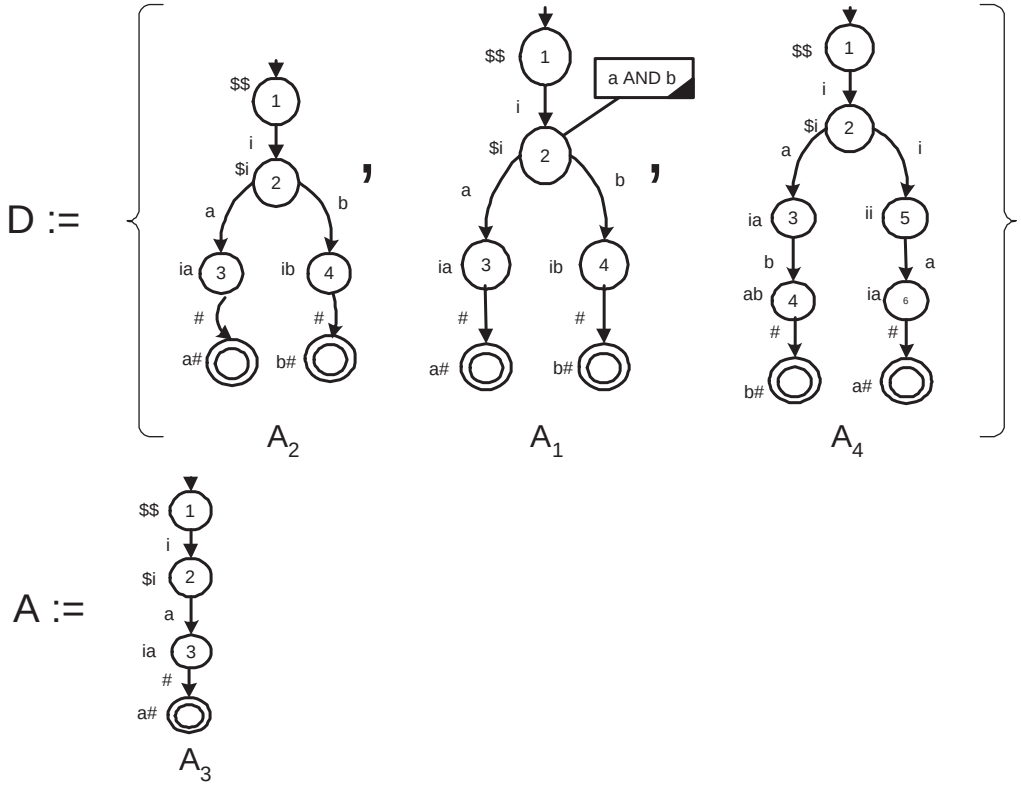


Figure 4.14: Evaluating Queries for Database Collection with Annotations

4.4.5 Searching for aFSAs with Complex Cycles

Most real-life business process specifications are based on a combination of descriptions with *no cycles* as well as those with *simple cycles* in their specifications, as exemplified by business process descriptions from RosettaNet and Open Travel Alliance [RosettaNet, 2005, OTA, 2003]. We believe that business processes with complex cycles are highly unlikely within the problem

Table 4.15: 2-gram table (t_1) for Figure 4.14

aFSA	2-gram (ℓ)
A_2	\$\$
A_1	\$\$
A_4	\$\$
A_2	\$i
A_1	\$i
A_4	\$i
A_2	ia
A_1	ia
A_4	ia

aFSA	2-gram (ℓ)
A_2	ib
A_1	ib
A_4	ib
A_4	ab
A_4	ii
A_2	*a*#
A_2	*b*#
A_1	*a*#
A_1	*b*#
A_4	*a*#
A_4	*b*#

Table 4.16: Annotations table (t_2) for Database Collection with Annotations

aFSA	2-gram (ℓ)	annotation (g)
A_2	\$\$	0000 0100 $\{i\}$
A_1	\$\$	0000 0100 $\{i\}$
A_4	\$\$	0000 0100 $\{i\}$
A_2	\$i	0000 0001 $\{a\}$
A_2	\$i	0000 0010 $\{b\}$
A_1	\$i	0000 0011 $\{a,b\}$
A_4	\$i	0000 0001 $\{a\}$
A_4	\$i	0000 0100 $\{i\}$
A_4	ia	0000 0010 $\{b\}$
A_4	ii	0000 0001 $\{a\}$

domain that is the focus of this dissertation. Thus evaluations that were carried out as part of this research were based on aFSAs with simple cycles. Thus, we leave aFSAs with complex cycles as future work. One approach for indexing aFSAs with complex cycles is to use a *hybrid search* approach, by combining the index search approach described in this research with sequential scanning, where the index search is used to first filter the results, followed by an application of sequential scanning on the intermediary results. A low value of look-back n is used for the filtering phase to minimize the exponential behavior of constructing the index, but this will also result in a large number of false matches resulting in high merge and query times. It is not possible to increase the look-back significantly to reduce the high false matches because of the high computational complexity of performing such an operation. Thus, a different approach is needed to deal with this problem. However, we believe that the approach described in this dissertation is sufficient for the business process matchmaking domain, where complex cycles are highly unrealistic.

Table 4.17: Query Evaluation on Annotated Database Collection

State	Val	v	intermediate result	final results
1	$\emptyset$	$\emptyset$	$\emptyset$	
2	$\{A_2, A_1, A_4\}$	$\{ \quad < a \rightarrow \{A_1, A_2, A_4\} >\}$	$\{A_2, A_1, A_4\}$	$\{A_2, A_1, A_4\}$
3	$\{A_2, A_1, A_4\}$	$\emptyset$	$\{A_2, A_1, A_4\}$	

4.5 Summary

Business process matchmaking is performed by formalizing business processes as aFSAs and checking for non-emptiness of the intersection aFSA. The intersection operation and emptiness checking are however not supported by traditional indexes as found in existing commercial databases. The only way to realize them is to build an abstraction layer above the index, to support intersection and emptiness-test operations. To find matching business process, the query aFSA is compared, via the abstraction layer, with each aFSA that is stored in the database. This means the entire database collection must be sequentially scanned, and each aFSA from the database be intersected with the query aFSA, the result of which is checked for emptiness-test. The following problems exist with this approach: i) sequentially scanning a large repository, for example, as is likely to be found in real life applications is inefficient, hence such an approach will not scale and ii) the complexity of computing intersection and checking for emptiness of the intersection aFSA is more than quadratic in effort and space utilization. Thus we need indexing support to match aFSAs. The criteria for two aFSAs to match is that they have at least a message sequence in common and the mandatory parts of the business process descriptions are fulfilled. Based on this observation, an indexing mechanism for matching business processes has been presented in this chapter. The indexing mechanism relies on two index structures: i) index structure based on message sequences and ii) index structure based on annotations on aFSA states. Indexing message sequences of an aFSA is complicated by the fact that the number of message sequences can be infinite due to cycles in the business process specification. In this chapter we have proposed an abstraction mechanism to resolve cycles in the aFSAs, and create an index based on this abstraction. The new index can be implemented using existing indexing data structures such as B^+-trees. We have defined three requirements for the resulting index which ensure that matching aFSAs are guaranteed to be found, the number of false matches introduced by the abstraction is minimized and the search space is reduced. As part of the indexing mechanism, a structure for indexing and evaluating annotations which are logical expressions associated to states has been presented also. The logical expressions are represented using fixed-length bit vectors and bitwise operations are used for computing set intersection, union and complement. The chapter also presented an algorithm for querying using the indexing structure to support matchmaking queries. Examples have been used to illustrate how the approach works. We have shown that the indexing mechanism proposed in the chapter meets all the three requirements of minimizing false matches, reducing the search space and guaranteeing that matches will be

found when they exist. An analysis of the indexing approach shows good computational properties for constructing the index and searching for aFSAs with simple cycles, being linear on the size of the database.

5 Implementation and Evaluation

This chapter describes an implementation and evaluation of the indexing and matchmaking approach that has been presented in Chapter 4 of this dissertation. The application domain on which the implementation is based is Web services discovery, with a focus on business process enriched services. The chapter comprises two sections which are the implementation section and the evaluation section. The implementation section presents a Web service-based architecture for matching Web services based on Universal Description, Discovery and Integration (UDDI) categories and business process descriptions. Input and output data to the system is also described as well as a tool for generating business processes using the RosettaNet standard. The evaluation section describes the goals of the evaluation as well as a description of the used data set. The environment of the experiments is also highlighted in the section, to be followed by results and their analysis. All evaluations were carried out based on aFSAs with simple cycles. The evaluation shows that the indexing and matchmaking approach presented in the dissertation allow all matching business processes to be found, without false misses. The results also show that the indexing approach performs better than sequential scanning approach by an order of magnitude. The results also show that the indexing mechanism scales well with an increasing data set collection, following a linear behavior. The index also performs better with increasing data set size, when compared to sequential scanning. Experimental results also confirmed the theoretical formulas presented in the preceding chapter.

5.1 Implementation

The indexing approach described in this dissertation was implemented and integrated into the IPSI Process Finder (IPSI-PF) engine. IPSI-PF is a service discovery engine extending the current UDDI infrastructure with process-aware service discovery capability [Wombacher et al., 2005, Wombacher et al., 2004c].

5.1.1 Architecture

A logical architecture of the implementation is shown in Figure 5.1. The **query formulation** component takes the query which comprises two parts: i) UDDI part which accepts the usual UDDI parameters [Ariba et al., 2000] and ii) the business process model, representing the associated business process to be used for matching. Both parts of the query are send through the communication layer to the **query decomposition** component. The **query decomposition** decomposes the query into the UDDI specific part and business process model specific part. The

UDDI query part is specified according to the UDDI API specification and the business process part is described using a standard such as BPEL4WS. The UDDI engine is responsible for processing the UDDI query part and the index-based process matching engine is responsible for processing the business process part.

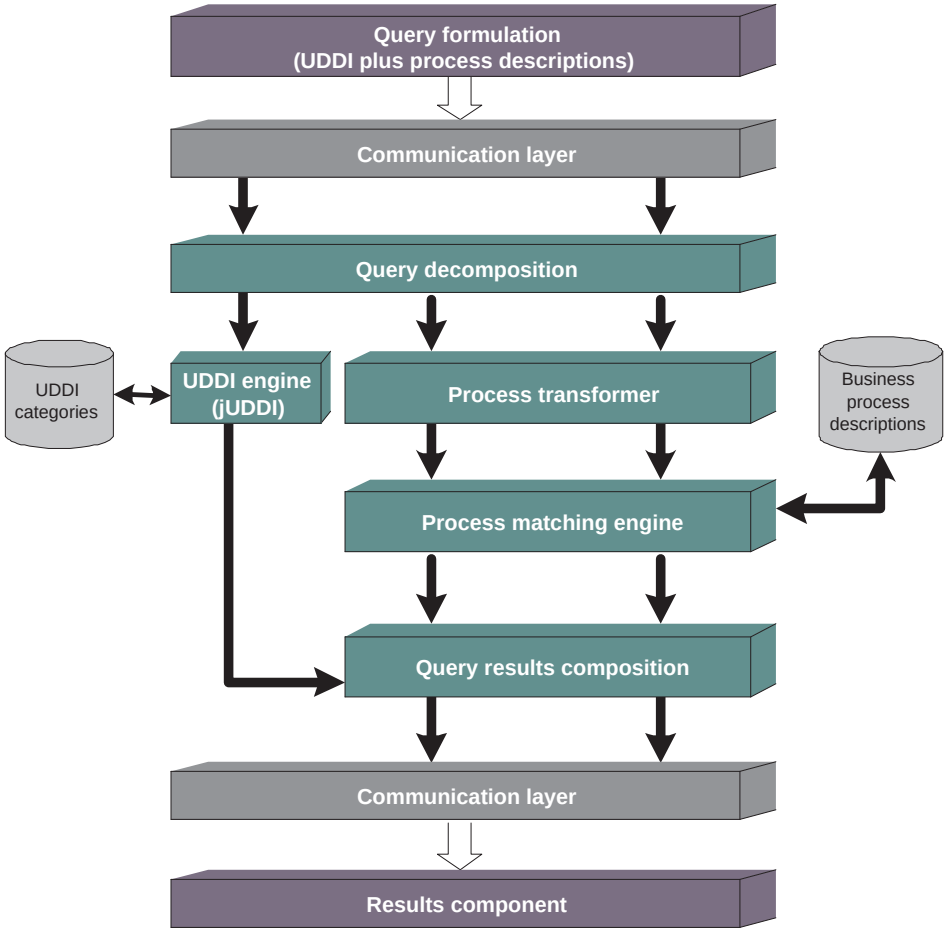


Figure 5.1: Logical Architecture

The UDDI engine used in the implementation is the open source Java implementation of the UDDI specification for Web services [JUDDI, 2003]. The **process transformer** component is responsible for transforming the input business process to the appropriate model like annotated finite state automata. The *index-based process matching engine* is implemented based on the indexing mechanism described in this dissertation. This engine is responsible for creating or setting-up the index, inserting business processes into the database, querying and performing maintenance operations on the underlying annotated finite state automata. As shown in Figure

5.1, there are two repositories, based on the standard relational database technology. There is a repository for storing UDDI specific information like categories according to predefined industry taxonomies. This allows standard UDDI queries to be performed. The other repository is the business process specific repository and is used by the index-based and sequential-scan based process matchers. Search results from the UDDI engine must be merged with results from the process matcher. This is achieved by ensuring that each business process in the business process repository is associated with a business service in the UDDI repository via a business service Universal Unique IDentifier (UUID) used as a foreign key. This foreign key allows the merging of partial results from the UDDI engine and process matcher. The final results are obtained by performing a natural join on the results from the two components using XSLT. The component responsible for these operations is the *query results composition* component. Finally results are collected and presented as a list via the *results component*.

This work did not focus on the UDDI part of this query, because it is standard and well-understood. Throughout the next sections, we thus focus on the business process part only. The next sections describe the input and output data for the process matchmaking components.

The input data described here refers to the input to the business process matcher. The input is a BPEL4WS document encoded in XML format. The document represents the business process of a service provider. The BPEL4WS is transformed to an aFSA which is the model for representing business process data for matchmaking as described in this dissertation. There exists tools to support the transformation of BPEL4WS documents to aFSA, for example [Wombacher et al., 2004b].

Figure 5.2 shows an example input data that is used by the process matcher. The figure shows an aFSA represented using XML format. The root element is an *automaton* element with three attributes describing the aFSA initial state, aFSA name and a UUID referencing a business service. The UUID key is also used in the aFSAs in the database collection to merge results of the UDDI search with those of process matcher search. The automaton element has four child elements, namely *roles*, *states*, *messages* and *transitions* elements. Children of the *roles* element represent the roles to which the business process belongs. In the example figure, the business process belongs to the *Buyer* role. Children of the *states* element comprise the states of the aFSA and each state has attributes describing whether the state is a final state or not, as well as the name of the state. Logical expressions are expressed as the children of each state. Only logical expressions denoting mandatory messages are explicitly represented in the example using an *AND* element. The messages element has *message* elements as children. Each message element has an attribute describing the name of the message. The messages are encoded as *sender#receiver#messageName*, with *sender* representing the role that sends the message, *receiver*, represents the role that receives the message and *messageName* represents the name of the message. In the example, a buyer process is represented and the bilateral collaboration partner is the seller. The transitions element has *transition* elements as children. Each transition element has attributes describing the source state, message label of transition and target state. The source and target states are defined in the *state* element.

The results obtained after submitting a query to the process matcher comprises a list of match-

```
<?xml version="1.0" encoding="UTF-8"?>
<automaton initialStateName ="s0" name ="3A_d4_32 " uuid = "bf3d12a4-a6e8-4ef2-918c-18c60a04edfd ">
    <roles >
        <role >Buyer </role >
    </roles >
    <states >
        <state isFinal ="false" name ="s0">
            <and >
                <var messageName ="Buyer#Seller#QuoteRequest "/>
                <var messageName ="Buyer#Seller#PriceandAvailabilityRequest "/>
            </and >
        </state >
        <state isFinal ="false" name ="s1"/>
        <state isFinal ="false" name ="s2">
            <and >
                <var messageName ="Buyer#Seller#PriceandAvailabilityRequest "/>
                <var messageName ="Buyer#Seller#QuoteAcknowledgementNotification "/>
            </and >
        </state >
        <state isFinal ="false" name ="s3"/>
        <state isFinal ="true" name ="s4"/>
        <state isFinal ="false" name ="s5"/>
        <state isFinal ="false" name ="s6"/>
        <state isFinal ="true" name ="s7"/>
        <state isFinal ="false" name ="s8"/>
        <state isFinal ="true" name ="s9"/>
    </states >
    <messages >
        <message name ="Seller#Buyer#PurchaseOrderConfirmation "/>
        <message name ="Buyer#Seller#PurchaseOrderRequest "/>
        <message name ="Seller#Buyer#PriceandAvailabilityResponse "/>
        <message name ="Seller#Buyer#QuoteConfirmation "/>
        <message name ="Buyer#Seller#QuoteRequest "/>
        <message name ="Buyer#Seller#PriceandAvailabilityRequest "/>
        <message name ="Buyer#Seller#QuoteAcknowledgementNotification "/>
    </messages >
    <transitions >
        <transition messageName ="Buyer#Seller#QuoteRequest " sourceStateName ="s0" targetStateName ="s1"/>
        <transition messageName ="Seller#Buyer#QuoteConfirmation " sourceStateName ="s1" targetStateName ="s2"/>
        <transition messageName ="Buyer#Seller#PriceandAvailabilityRequest " sourceStateName ="s2" targetStateName ="s3"/>
        <transition messageName ="Seller#Buyer#PriceandAvailabilityResponse " sourceStateName ="s3" targetStateName ="s4"/>
        <transition messageName ="Buyer#Seller#PurchaseOrderRequest " sourceStateName ="s4" targetStateName ="s5"/>
        <transition messageName ="Seller#Buyer#PurchaseOrderConfirmation " sourceStateName ="s5" targetStateName ="s0"/>
        <transition messageName ="Buyer#Seller#QuoteAcknowledgementNotification " sourceStateName ="s2" targetStateName ="s9"/>
        <transition messageName ="Buyer#Seller#PriceandAvailabilityRequest " sourceStateName ="s0" targetStateName ="s6"/>
        <transition messageName ="Seller#Buyer#PriceandAvailabilityResponse " sourceStateName ="s6" targetStateName ="s7"/>
        <transition messageName ="Buyer#Seller#QuoteRequest " sourceStateName ="s7" targetStateName ="s8"/>
        <transition messageName ="Seller#Buyer#QuoteConfirmation " sourceStateName ="s8" targetStateName ="s4"/>
    </transitions >
</automaton >
```

Figure 5.2: Example Input Data for Buyer Business Process

ing aFSAs representing business processes, where each aFSA contains a UUID key for the business service it represents. The aFSAs and associated UUIDs are used to perform merge operations with results from the UDDI engine using natural join operation .

5.1.2 Data Generation

Due to the fact that we could not find real world service descriptions providing a process descriptions, we developed a tool to generate and categorize RosettaNet Partner Interface Processes (PIPs) [RosettaNet, 2005] into complex business processes represented as aFSAs.

The RosettaNet standard specifies semantics for messages to be used for describing business processes. The messages are compiled into a data dictionary. In addition, RosettaNet also

```
<?xml version="1.0" encoding="UTF-8"?>
<rules segment ="3A">
    <pip id="3A1" startsSequence ="yes" selfCycle ="yes">
        <messages >
            <message >Buyer#Seller#QuoteRequest </message >
            <message >Seller#Buyer#QuoteConfirmation </message >
        </messages >
        <preconditions >
            <id type ="negative ">3A4 </id>
        </preconditions >
    </pip>
    <pip id="3A2" startsSequence ="yes" selfCycle ="yes">
        <messages >
            <message >Buyer#Seller#PriceandAvailabilityRequest </message >
            <message >Seller#Buyer#PriceandAvailabilityResponse </message >
        </messages >
        <preconditions >
            <id type ="negative ">3A4 </id>
        </preconditions >
    </pip>
    <pip id="3A4" startsSequence ="yes" selfCycle ="no">
        <messages >
            <message >Buyer#Seller#PurchaseOrderRequest </message >
            <message >Seller#Buyer#PurchaseOrderConfirmation </message >
        </messages >
        <preconditions >
            <id type ="negative ">3A4 </id>
        </preconditions >
    </pip>
</rules>
```

Figure 5.3: Partial Rules File for RosettaNet Cluster 3, Segment A

specifies rules for combining atomic business processes called PIPs into more complex ones. The data generation and categorization tool has two major components (i) sequence generator and (ii) aFSA process generator.

Sequence Generator

The sequence generator part of the tool takes a RosettaNet rules file, and uses it to construct sequences of RosettaNet messages, which comply to the PIP specifications. The *rules file* is an XML file representing PIP specifications. Each PIP is described by several properties most of which are taken from the RosettaNet specification. Other properties are not explicitly given in the PIP specification and must be derived by analyzing the PIP. An example of a property that is not given is whether or not executing a PIP gives rise to a self-cycle. Properties taken from the specification include PIP id, message names, role names and pre-conditions for executing the PIP. Figure 5.3 shows a snap shot of a partial rules file for Order Management (cluster 3), and in particular the Quote and Order Entry segment (segment A) of the RosettaNet PIP specification. The figure shows how the PIP properties described above are represented in XML format.

The sequence generator accepts as input a file like that shown in Figure 5.3 and uses it to generate sequences. The algorithm checks for all PIPs that can be combined and it generates sequences of up to a maximum depth. The depth is a parameter specified on the tool to limit the

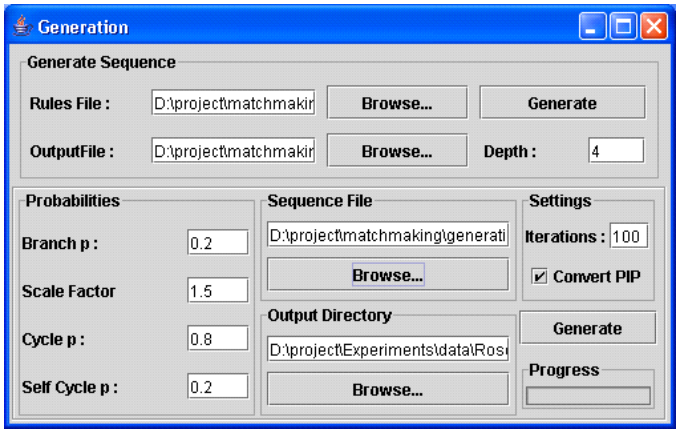


Figure 5.4: Graphical User Interface for a Configurable Data Generator Tool

maximum depth of each sequence. Sequences constructed from the sequence generator are used by the FSA process generator to construct aFSAs.

aFSA Process Generator

The aFSA process generator tool can be configured to construct aFSAs from sequences generated by the sequence generator by randomly selecting a sequence, iterating over the sequence and determining sequences with the same prefix and adding the additional sequence as part of the process dependent on a number of parameters. Figure 5.4 is a GUI for the process generator tool showing the different parameter options that can be configured. Branch probability controls branching in aFSAs being generated. Higher branch probability values result in a higher transition fan out at every state. This tends to increase the complexity of aFSAs. The scale factor determines how fast or slow the process graph grows from the start state. Cycle occurrence is controlled by the cycle probability while self-cycle occurrence is controlled by the self-cycle probability. High values for the cycle and self-cycle probabilities result in highly cyclic process specifications, which can be quite complex. The number of iterations determines the number of aFSAs to be generated from the sequences.

aFSAs are constructed by combining sequences from the sequence file generated from the sequence generator. Care is taken to ensure that each resulting structure is a valid aFSA, with a single start state, and one or more final states and that it is a connected graph. Using the sequence and aFSA process generators, we can generate aFSAs from RosettaNet PIP specifications, and analyze the different categories of data according to the parameters described above. We can also measure the performances of the different categories of data.

5.2 Experimental Evaluation

This section describes evaluations that were carried out as part of this research. First, the goals of the evaluations will be described, followed by an overview of the environment of the experiments. The data set used for the experiments is described as well as results and their analysis. The results show that the indexing mechanism performs better than sequential scanning by a significant order of magnitude. Also the indexing mechanism performs better with increasing data set size, when compared to sequential scanning. Another important result of this evaluation is that the index searching mechanism scales well with increasing data set size.

5.2.1 Experimental Evaluation Goals

The goal of the experimental evaluation was to characterize indexed search complexity, performance and quality of search results. In particular, we wanted to find out the following:

- performance of index-based search versus sequential search,
- influence of data-set size on index performance,
- influence of look-back on index performance,
- influence of look-back on quality of search results,
- conformance of theoretical results to experimental results.

The performance of index-based search is expected to be better than that of sequential search, especially for large data sets. The reason for this is that sequential scan search depends on sequentially scanning through the entire data collection, comparing each aFSA from the database with the query aFSA; on the other hand index-based search only searches a limited number of aFSAs from the database. This results in better query performance for the index-based search. The experiments will determine by what factor index-based search is better than sequential search. Other measurements that help us to better understand the index are the effect of look-back on performance and quality of search results. We also want to determine how close theoretical formulas conform to experimental results.

5.2.2 Environment of the Experiments

The baseline infrastructure is sequential search-based matchmaking engine that scans the data set, computes intersection with the query aFSA and checks for non-emptiness of intersection results. An indexing engine is implemented at the same level of abstraction, based on the approach described in this dissertation. The experiments were conducted on a Dell machine, with a Pentium 4 processor 2.00GHz clock speed and 1000 MB RAM. The total disk space was 74 GB. The machine was running under Windows XP operating system. MySQL server version 4.0 was the used database engine. The machine was also running the JBOSS 3.2.3 application

server which provided the J2EE environment for the IPSI-PF process matchmaking engine. The sequential and index search are implemented on the same data model and level of abstraction using container managed persistency. We allowed no buffering/caching of query results; all tests were run under cold start conditions.

Data Set

The data-set used for the experiments is business process data based on the RosettaNet specification [RosettaNet, 2005]. The RosettaNet data set ensures a realistic data-set with a significant level of complexity.

Table 5.1: aFSAs with Simple Cycles Structural Complexity

N	$\|Q\|$	$\|\Sigma\|$	$\|\Delta\|$
100	7 078	1 563	7 385
200	14 156	3 126	14 770
400	28 312	6 252	29 540
600	42 468	9 378	44 310

The data generator tool, which is described in an earlier section was used to generate complex business processes from the RosettaNet PIPs. RosettaNet PIPs were composed into more complex sequences according to RosettaNet rules for combining PIPs. Parameters such as branching, cycle and self-cycle probabilities as well as business process graph depth were used to configure the tool for data generation.

Table 5.2: Data Set/ Look-back Matrix

$N\backslash n$	0	1	2	3	4
100	1 663	4 443	6 403	7 682	8 607
200	3 326	8 886	12 806	15 364	17 214
400	6 652	17 772	21 500	30 728	34 428
600	9 978	26 658	38 418	46 092	51 642

The data was partitioned into four sets of sizes 100, 200, 400 and 600 business processes. The partitioning was to allow the measuring of query time as a function of data-set size. This measurement can be used to answer the question, whether the index search approach scales with large data sets. The look-back was varied from 0 through to 4, to measure the influence of look-back on index search performance, as well as finding out the influence of the look-back on the search quality - false match rate. The structural complexity of input business process, which are modeled as annotated finite state automata was also measured and recorded. The information recorded includes the number of states, messages, transitions and sequences. Table 5.1 summarizes the complexity of the business processes and Table 5.2 shows the number of sequences

recorded for each data set as the look-back n, was varied from 0 to 4. The used parameters in the two tables are as follows: N is the data set size, $|Q|$ is the number of state objects, $|\Sigma|$ is the number of message objects and $|\Delta|$ is the number of transition objects. Table 5.2 shows that generally, increasing the look-back results in an increase in the number of sequences. This is because for cyclic aFSAs, more iterations are needed as the look-back is increased. Thus, the result is more sequences - also referred to as n-grams throughout this dissertation are generated. Table 5.1 shows that when the data set is increased, the number of objects in the database for each $|Q|$, $|\Sigma|$ and $|\Delta|$ also increases almost linearly. The reason for this almost linear increase is that the structural complexity of the annotated finite state automata is not changing, since the same data is used, but what is increasing is the number of objects in the database.

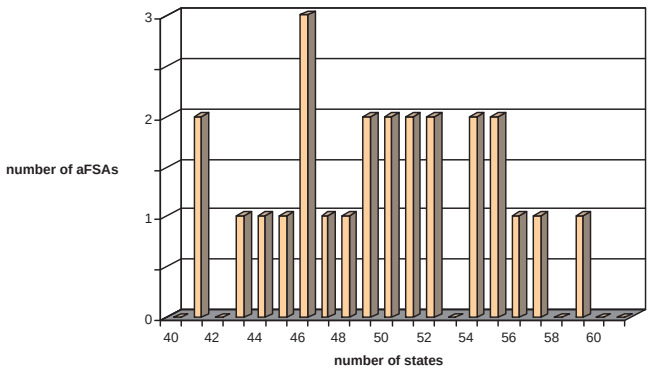


Figure 5.5: Query aFSAs Plotted against Number of States

The input query aFSAs were also analysed. The set of aFSAs used as the query was a set of twenty five (25) business processes modeled as aFSAs. Figure 5.5 shows query aFSAs plotted against the number of states. States in the query aFSAs can be characterized as follows: the number of states is distributed between forty and sixty, with more than 70% of the aFSAs having states between 46 and 56 states. The number of transitions in the query aFSAs was also measured. Thus the average number of states in each query aFSA is high.

Figure 5.6 shows a plot of query aFSAs against the number of transitions. The number of transitions in the query aFSAs lies between 40 and 62, with more than 80% of the query aFSAs having between 46 and 60 transitions each. This also indicates that the number of transitions in each query aFSA is significantly high.

Another measurement of interest for the query aFSA data set was the relationship between states and transitions. Figure 5.7 is a scatter plot showing transitions versus states in the query aFSA data set. The scatter plot shows that the number of transitions increases almost in direct proportion to the number of states in the query aFSAs. This behavior is understandable because

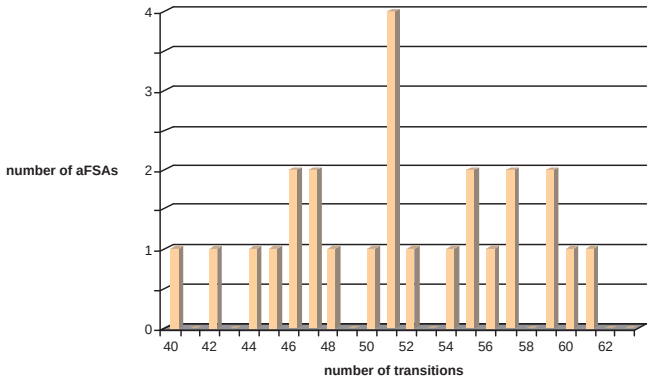


Figure 5.6: Query aFSAs Plotted against Number of Transitions

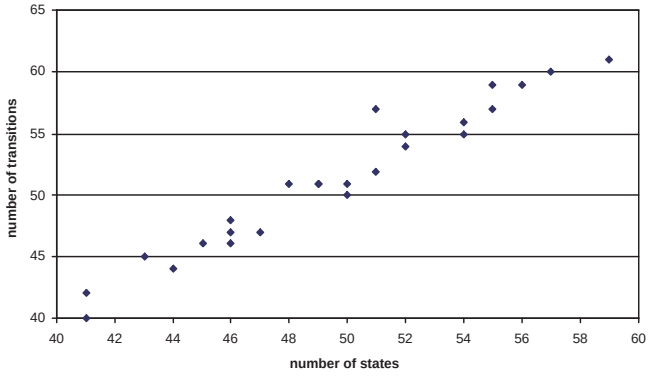


Figure 5.7: Query aFSAs - Number of Transitions Plotted against Number of States

within the aFSA graph, states are connected by transitions, thus the more states there are, the more transitions are needed to connect states to form a connected graph. Another behavior to observe from Figure 5.7 is that different numbers of transitions do associate with the same number of states. As an example, 46 states associate with 46, 47 and 48 transitions respectively showing that there is no one-to-one relationship between the number of states and the number of transitions. This is also understandable because different states associate with different numbers of transitions in an aFSA.

The last measurement on the query aFSA data set was with respect to cycles. Figure 5.8 shows

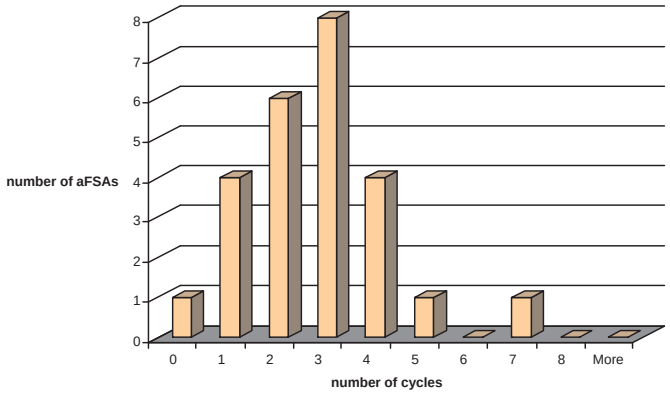


Figure 5.8: Query aFSAs - Number of aFSAs Plotted against Number of Cycles

the number of cycles versus the number of query aFSAs. The figure shows that almost 90% of the query aFSAs had between 1 and 4 cycles each and only one aFSA had no cycle. Thus the data provides the characteristics motivating the index described in this dissertation.

Evaluation Results

This section presents the experimental results and their analysis. The results are reported based on the following: the influence of data set size and look-back on index search performance, influence of look-back on quality of results as well as the conformance of some of the results to corresponding complexity formulas presented in Chapter 4. The results show a significant performance gain as compared to sequential scanning. The results also show that the index performance factor over sequential scanning increases as the data set size is increased. Results also show that the index search time varies linearly with the data set size, indicating that the index scales with the data set size. The number of false matches was very low as anticipated and the theoretical formulas were found to conform to the experimental results. No false misses were reported in all the experiments that were carried out. The results are discussed in more detail in the next subsections.

False Misses

No false misses were reported for the index search approach as anticipated. This result was found by comparing the output from sequential scanning and that from the index search, for the same data sets. We found that for every input aFSA, all the sequential scan results were also re-

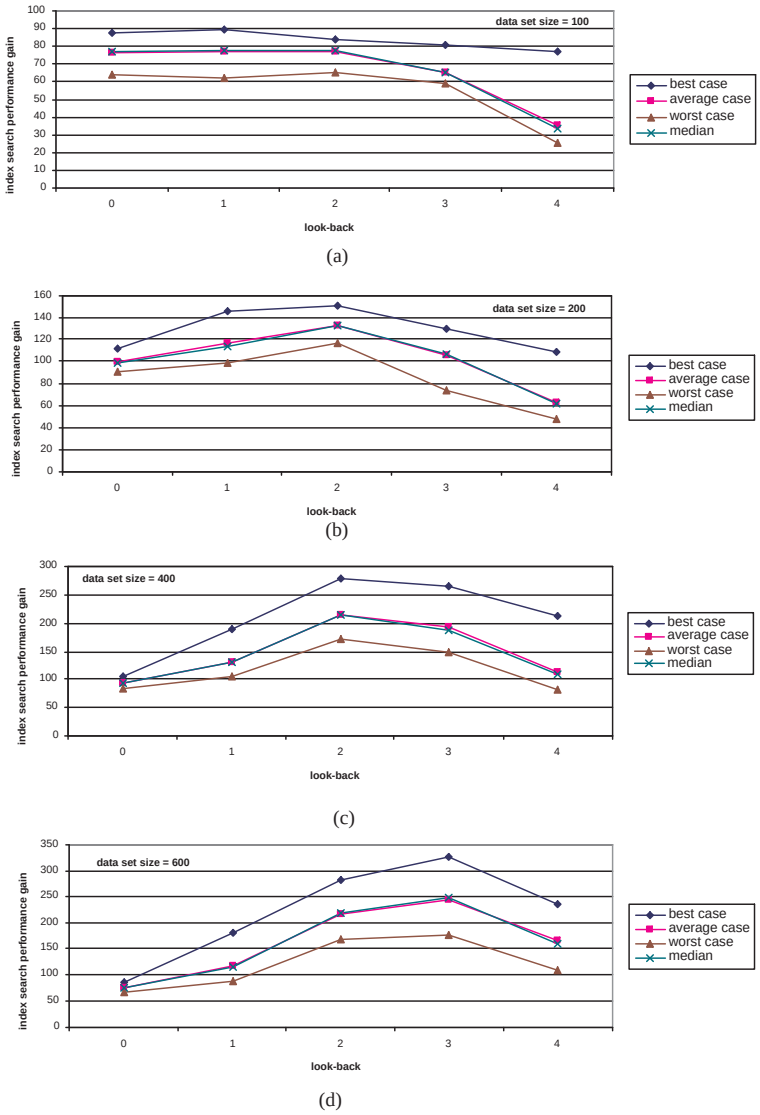


Figure 5.9: Performance Gain Factor over Sequential Scan for (a) 100 (b) 200 (c) 400 and (d) 600 aFSA Data Sets

turned by the index search. However, false matches were reported for the index search approach, the number of which being greater for lower look-backs. The rate of false matches was however reduced by increasing the look-back as described in the indexing approach.

Performance Results

Performance measurements were taken for each data set. For each data set, performance factors were obtained by dividing the sequential search time by the index search time, to give the factor by which index search is faster than sequential scan. From the performance factors, in each data set, the best case, average case, worst case and the mean case performance factors were measured and recorded as the look-back was varied from 0 to 4. Throughout this chapter, we use the term *worst case complex* to mean the complexity for aFSAs with simple cycles. Figure 5.9 (a) - (d) shows performance measurements for the 100, 200, 400 and 600 aFSA data sets as the look-back was varied respectively.

Figure 5.9 (a) shows results for the 100 aFSA data set. The figure shows that performance factors ranged from 25 to 89, with the 25 being the worst case performance factor and occurred when $look-back = 4$. The best case performance of 89 is when $look-back = 1$. The average and mean value performance factors ranged from 34 to 78 where lowest median of 34 occurred when $look-back = 4$ and the best median of 78 occurred when $look-back = 1$. The figure also shows that in general, performance factors tend to decrease when the look-back is increasing.

Figure 5.9 (b) represents results for the 200 aFSA data set - which is double that for Figure 5.9 (a). The results are as follows: performance factors ranged from 48 to 151, with the 48 being the worst case performance factor and occurred when $look-back = 4$. The best case performance of 151 occurred when $look-back = 2$. The average and mean value performance factors ranged from 62 to 132 with the lowest median of 62 occurring when $look-back = 4$ and the best median of 132 occurring when $look-back = 2$. A quick comparison of the 100 and 200 aFSA and data sets shows that the factors have almost doubled for the 200 aFSA data set. Another important result is that the best case performance factors have shifted for the 200 aFSA data set to occur when $look-back = 2$ as opposed to the 100 aFSA data set case, where best case performances occurred when $look-back = 1$.

Results for the 400 aFSA data set are shown in Figure 5.9 (c). The results for this data set are as follows: performance factors ranged from 81 to 280, with 81 being the worst case performance factor and occurring when $look-back = 4$ and 280 being the best case performance factor and occurring when look-back$= 2$. The average and mean value performance factors ranged from 92 to 215 with the lowest median of 92 occurring when look-back$= 0$ and the best case median of 215 occurring when look-back$= 2$. For the 100 and 200 aFSA data sets, the worst case values for the average and median values occurred when look-back$= 4$, but for the 400 data set, the worst case performance factor (for the average and median values) occurred when look-back$= 0$. This is also shown in the figure. The average and median performance factors also increase as the data size in increased as shown in the figures.

The last data set size, which was also the largest in the experiments was the 600 aFSA data set. The performance results are shown in Figure 5.9 (d). For this data set, performance factors ranged from 68 to 328, with 68 being the worst case performance factor which occurred when $look-back = 0$ and 328 being the best case performance factor which occurred when look-

back = 3. The average and mean value performance factors ranged from 76 to 246 with the lowest median of 76 occurring when look-back = 0 and the best case median of 246 occurring when look-back = 3. In this largest data set used for the experiment, worst case performance factors all occurred when look-back = 0 and best case performances occurred when look-back = 3. Thus in general, the index search performs better with an increased data set size.

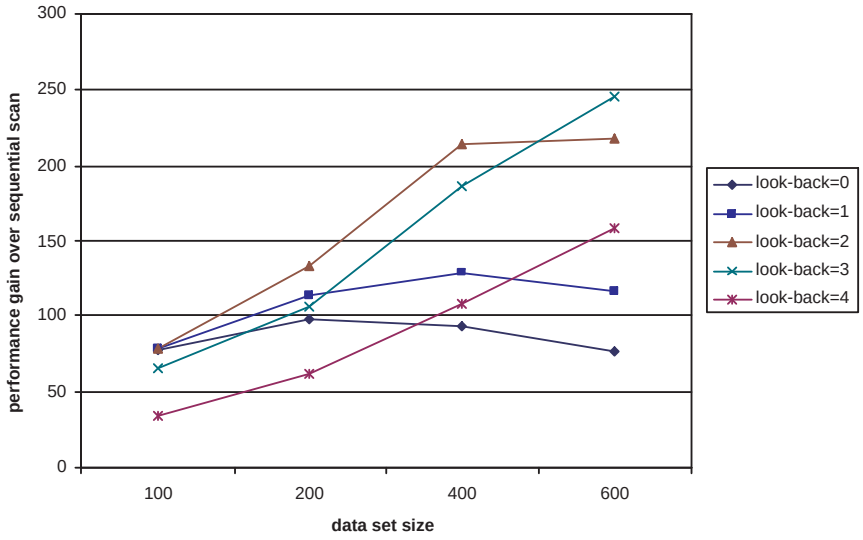


Figure 5.10: Performance Gain Factor over Sequential Scan versus Data Set Size

Figure 5.10 shows a more compact view of the index search performance results. The figure shows how the performance factors changed with different data set sizes for different look-backs, where only median performance factor values are used. The figure shows that the look-back of zero starts-off with a performance gain of 77 for the 100 aFSA data set which rises to 98 for the 200 aFSA data set before it starts falling to 93 and 76 for the 400 and 600 aFSA data sets respectively. The same behavior can be observed for the look-back of 2, which starts-off with a performance factor of 78 for the 100 aFSA data set, peaks at the 400 aFSA data set to a performance factor of 129, before falling 116 for the largest data set of 600 aFSAs. The look-back of 2 has overall increasing performance factor values, starting-off with a performance factor of 78, which rises to 133, 214 and 218 for the 100, 200, 400 and 600 aFSA data sets respectively.

The look-back of 3 and 4 have performance factors which are also increasing with increasing data set size. In particular, the look-back of 3 starts-off with a performance factor of 65 for the smallest data set of 100 aFSAs and rises to 106, 187 and 246 for the 200, 400 and 600 aFSA data sets respectively. The look-back of 4 starts-off with a performance factor of 34 for the 100 aFSA data set, rising to 62, 108 and 159 for the 200, 400 and 600 aFSA data sets respectively.

Performance Results Analysis

Performance results in Figure 5.9 and Figure 5.10 show that the index search performs much better than sequential scanning, with performance factors over sequential scanning ranging from a median of 34 to 246. The best median performance factor result of 246 occurred for the largest data set of 600 aFSAs and for a look-back of 3.

The explanation for this is that the effort to perform sequential scan search operations is much higher because the entire database collection must be scanned, and for each aFSA in the collection, expensive intersection operations must be performed. Thus, the larger the database collection, the more computation effort is required. The computation effort increases linearly as the database collection size increases, because in our experiments, constant queries were used, but the database size was varied. On the other hand, for the index search, only a subset of the database collection is searched because the search space is pruned by the index and query operations require logarithmic computational effort for the best case (see Section 4.4.3). This explains the significantly high and increasing performance factor margins between sequential scan and index search approaches.

The behavior of the curves with look-backs 0 and 1 respectively (Figure 5.10) where they peak and then fall as the data set size increases can be explained as follows: for look-backs of 0 and 1, the context information used in matchmaking operations is rather minimal (look-back=0 and look-back=1) resulting in a high number of false matches (see Figure 5.11). This means that the search space is large due to the high false match rate. We have already explained in Section 4.4.3 about the effect of false matches on search performance, where we pointed out that a high number of false matches increases the search space, thus can significantly affect the overall search performance. Since the data collection was increased by adding copies of the same data to keep the complexity of the data collection constant, the number of false matches was multiplied as the size of the data collection increased, thus multiplying the search space by the same factor. The result is a deterioration in the performance factor if the false matches keep on multiplying as shown in Figure 5.10 for look-backs of 0 and 1. The search space is reduced by increasing the look-back. In our experiments look-backs above 3 showed an increasing performance gain as shown in Figure 5.10.

The behavior we observe in Figure 5.10 for the look-backs of 0 and 1 should not occur in practice because multiple copies of the same data are not normally used as a data collection. It is however necessary in practice to understand the characteristics of the data collection in order to determine the most appropriate look-back to use where the performance is optimal and false match rate is minimum. It might be necessary to reindex the data collection whenever the performance (and false match rate) become unacceptable. We should be aware that increasing the look-back also increases the complexity of searching because the number of n-grams has increased (see Table 5.2). This explains why the performance factors are lower as the look-back was increased. So for a given data collection, there is an optimal value for the look-back that maximizes the performance as well as results quality. For example, as described in the next

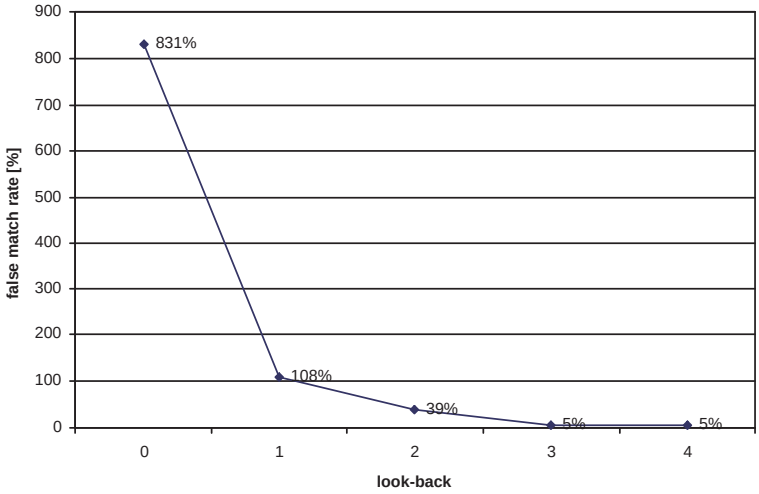


Figure 5.11: False Match Rate

sub-section, in the data collection used for the experiments, a 5% false match rate was obtained when look-backs of 3 and 4 were used. So a performance factor of 246 gives a good factor as well as a good false match rate (5%).

False Matches

We now discuss false match results. The rate of false matches is important because if the false match rate is too high, the index search results are not of much practical use. Moreover, the number of false matches also have an impact on the search performance as described in Section 4.4.3 and also in the paragraphs above. Figure 5.11 shows how the false match rate varies with the look-back. With a look-back of zero, the false match rate was rather high at 831%. When the look-back was increased to one, the false match rate dropped to 108%; for a look-back of 2 it dropped to 39% before stabilizing at 5% level for look-backs of three and four.

Analysis of False Match Results

The false match results are consistent with the theoretical analysis. When look-back is zero, no context information is being taken into account. This means that individual messages are being compared during query evaluation, thus chances that a match is found are rather high. Increasing the look-back reduces the ambiguity of which sequences are matching because more context information is taken into account when matching as described in Chapter 4.

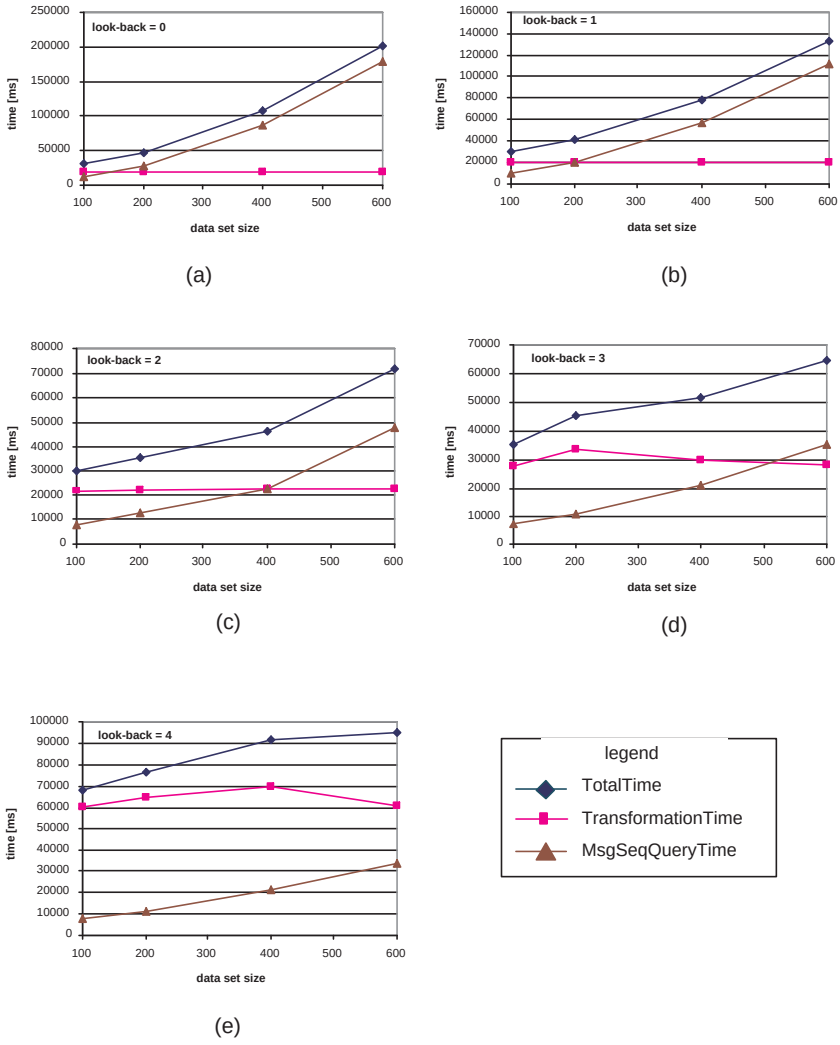


Figure 5.12: Time versus Data Collection Size

Time versus Data Collection Size

We were also interested in finding out how the search time, query construction time and total query time vary with the data collection size. These measurements are important because they help us to determine if the index is scalable or not, as the data collection size increases. To this

end we have done some experiments to measure these values. The results are shown in Figure 5.12. In Figure 5.12 (a) results for a look-back of 0 are shown. The transformation time, which constitutes the bulk of the query construction time is constant for the different data collection sizes. The message query time, which is the time to query using the message sequence index, and the total query times increase linearly as the data collection size increases for look-backs 3 and 4, but more than linear for look-backs 0 and 1. The same data collection (for the query) was used for the different look-backs, hence the time needed for construction is constant. The linear time, as opposed to the expected logarithmic time (see Equation 4.28) can be attributed to the fact that aFSAs with simple cycles were used in the experiments, rather than aFSAs without cycles, thus increasing the number of n-grams to be indexed and searched significantly. Be aware that the exponential complexity for searching is with respect to the number of transitions in the most complex cycles of aFSAs. The fact that this number is usually small explains the linear, rather than exponential behavior. This behavior is shown in Figure 5.12 (a) - (c) for look-backs of 0 to 2 respectively. The transformation time is supposed to be constant because the same queries are being used for querying the different data sets. The constant behavior can be seen in Figure 5.12 (a) - (c), changing slightly for Figure 5.12 (d) - (e).

Total Query Time versus Data Collection Size Analysis

The search time shown in Figure 5.12 (a) - (c) for look-backs of 0 - 2 respectively is more than linear due to the high rate of false matches, being 831%, 108% and 35% respectively (see Figure 5.11). Like we pointed out already in Section 4.4.3, a high number of false matches can affect search performance in a significant way, by increasing the search time since the search space is large. Regression analysis on search time for look-backs above 3 confirmed a linear relationship with the data collection. The best fit curve obtained from regression analysis was $f(x) = a*x+b*\mathrm{Log}(x)+c$ where a, b and c are constants and x is an independent variable representing the data collection size and the function f(x) represents the total search time. The margin of error was below 1%. Theoretical results for the total search complexity were also compared with experimental results. Experimental results were found to be in conformance with theoretical complexity results described in Chapter 4. The following conclusions can be drawn from these results: the index search approach scales with an increasing data collection size for a given look-back n.

Theoretical Analysis versus Experimental Results

In this sub-section we verify the aFSA transformation formula given in Chapter 4 with experimental results. The computational complexity for constructing the message sequence index was given as $O\left(|\Delta_q|\right)$ for the best case and $O\left(|\Delta'_q|^{n+1}\right)$ for the worst case, respectively, where $|\Delta'_q|$ is the average number of transitions in the most complex cycle of the query aFSA and n is the look-back. The aFSA with simple cycles complexity formula for transforming aFSAs

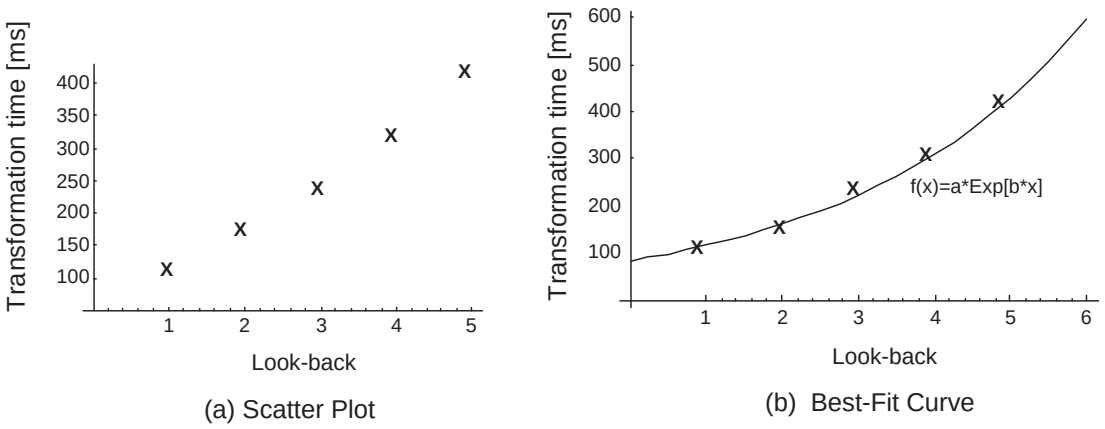


Figure 5.13: (a) Scatter Plot (b) Best-Fit Curve

was experimentally verified. The formula for transforming a query aFSA can be represented as $f(|\Delta'_q|, n) = |\Delta'_q|^{n+1}$. In the experiment, the most complex query aFSA was used for the evaluation. $|\Delta'_q|$ was made constant while n was varied. The transformation time in milliseconds was measured as the look-back n varied. When $|\Delta'_q|$ was constant, the theoretical formula corresponded to the model: $f(n) = a^{n+1}$ with a being a constant. The function $f(n) = a^{n+1}$ is exponential on the look-back n.

Using regression analysis the best-fit curve was $f(x) = a^x$ with a margin of error of 0.30% where x corresponds to the look-back n and the function $f(x)$ corresponds to $f(n)$, the transformation time, was obtained.

Figure 5.13 shows a scatter plot of transformation time (in milliseconds) versus look-back and an exponential best-fit curve. In the figure, crosses represent values recorded from experiments where the most complex aFSA was used, while the curve is a best-fit curve. To be certain, about the validity of our formula, we also took measurements for the transformation time when $|\Delta'_q|$ was varied and n kept constant. The formula became $f(|\Delta'_q|) = |\Delta'_q|^{n+1}$ which is a polynomial function. Using regression analysis we were able to confirm that the best-fit curve was a polynomial curve, with power corresponding to $n+1$ and a margin of error around 0.50%.

The best-fit for the sort-merge formula was a quadratic curve with an error margin of 2.21% which does not conform to an $O(N' * logN')$ that we presented. One explanation for this difference is the way in which results were merged in the implementation which has not been optimized for merge-sorting to realize the $O(N' * logN')$ complexity we had assumed. However, from our experiments, the merge time was very low (less than 1 second) as compared to other times such as, transformation time for the message sequence index. Most of the search time complexity came from actual searching and query construction.

5.3 Summary

The chapter presented an architecture for an index-based matchmaking engine as well as an evaluation of the indexing approach for business process matching within the Web service domain. The architecture illustrated how the indexing approach can be used to enrich the current UDDI service discovery infrastructure with business process matchmaking capability. The chapter also described detailed evaluations for the indexing approach using business process data derived from RosettaNet PIPs. The goals of the evaluation included comparing index search with sequential scanning, finding out the effect of data collection size on index performance, the effect of look-back on performance and quality of results, and a comparison of theoretical results with experimental results.

The index search approach performed better than sequential scanning by a significant order of magnitude, with median factors between 34 and 246 for the worst and best case performance factors respectively. The best median performance factor occurred for the largest data collection of 600 aFSAs and for a look-back of 3. The explanation for this is that the effort to perform sequential scan search operations rises by at least a quadratic factor (complexity to do intersection) to the data collection size, each time the data collection size increases, while for the index search, it rises linearly to the size of the database for a given look-back n. Thus as the database size increases, the time needed to perform sequential scan operations increases much faster than that to perform index search, which increases more slowly. Also, increasing the look-back reduces the number of false matches, thus reducing the search space for the index and increasing the quality of search results. At the same time, increasing the look-back also increases the complexity of searching because the number of n-grams increases as shown by the input data analysis (see Table 5.2), explaining why the performance factor drops for high look-backs. So for a given data collection, there is an optimal value for the look-back that maximizes the performance as well as the quality of results. This look-back was 3 in our experiments with a 5% false match rate.

Regression analysis also showed that the index search time increases linearly, with an increase in the data collection size for a look-back above 2 for our data collection. This means that the index search approach scales with an increasing data collection size for a given look-back and data collection. Regression analysis was also done to check the conformance of some theoretical results to experimental results by using curve fitting techniques and nonlinear regression calculations to check for standard errors. Theoretical results conformed to experimental results within margins of errors less than 1%.

There were no false misses in the experimental results as anticipated. False matches occurred and were reduced from a high value of 831% to a mere 5% after increasing the look-back from 0 to 3 respectively. At a false match of 5%, we were able to attain the highest performance factor of 246 over sequential scanning as already stated.

6 Conclusions and Future Work

6.1 Achievements of Dissertation

The main contributions of this dissertation are as follows: (i) an indexing technique for efficiently querying business processes in large repositories was developed and formalized and (ii) A Web service matchmaking engine for business processes that was based on the indexing approach was implemented and experimentally evaluated; the results of the evaluation showed that the index-based search outperformed sequential scanning by a significant order of magnitude.

6.1.1 Indexing Techniques for Matching Business Processes

The formal definition of matchmaking of business processes is based on computing the intersection of aFSAs and checking the intersection aFSA for non-emptiness. We have shown in the dissertation that computing intersection of aFSAs and checking the intersection aFSA for non-emptiness requires more than quadratic computation effort. Even if querying individual aFSAs had low computation complexity, sequentially scanning Internet scale repositories to compare individual business processes for a match is impractical without indexing support. To this extend, an indexing technique to support efficient matchmaking of business processes in large service repositories was developed in this dissertation. The indexing approach is based on two indexes: (i) a message sequence index for indexing message sequences and (ii) an annotation index for indexing logical annotations. The message sequence index relies on the language of aFSAs to build an index, which is used for searching based on message sequence equivalence. Unfortunately the number of message sequences is potentially infinite due to cycles in business process specifications. We have developed an abstraction mechanism that reduces the aFSA language to a finite language by resolving cycles. The finite language is used for building the index. The abstraction mechanism ensures that (i) no false miss will occur during search operations (ii) the number of false matches introduced by the loss of information (abstraction mechanism) is minimized and (iii) search performance is increased. We have shown that the new indexing mechanism developed in this dissertation meets the three stated requirements.

The annotation index is used to index annotations, which are propositional logic expressions associated to aFSA states. These expressions must be evaluated when querying business process descriptions. The annotation index uses a bit vector index mechanism to index propositional logic expressions, and relies on fast bit operators of OR, AND and $COMPLEMENT$ to express

set union, set intersection and set complement operations. A complex algorithm for querying the two indexes to find matching business process descriptions was also described. The complexities of the indexing mechanisms and the search algorithm were analyzed and described in the dissertation. The indexing approach has good search complexity, being approximately linear on the number of aFSAs to be searched.

6.1.2 Web Service Matchmaking Engine for Business Processes

As proof of concept the matchmaking approach and indexing mechanism for business process matchmaking were implemented as part of the IPSI-PF business process matchmaking engine project. The engine extends the UDDI service discovery mechanism with process semantics, thus allowing discovery based on business process descriptions. A run-time architecture showing how our approach was incorporated into the service discovery infrastructure of Web services was given and described. Performance evaluations were carried out using business process data generated from RosettaNet PIPs. Performance evaluation results showed that the indexing approach outperformed sequential scanning by several orders of magnitude. The results also showed that the index performance improved with increasing data set size, thus our index becomes more relevant for large data sets, which are anticipated in large service discovery repositories on the Internet. We also carried out regression analysis to determine the behavior of the indexing approach with increasing data sets, using our experimental results. Regression analysis of these results showed that search time increased linearly to the increase in data set size, meaning that the indexing approach scales with increasing data set sizes.

6.1.3 Application of the Approach to Other Domains

The indexing approach (or its specialization) proposed in this dissertation can be applied to other application domains. In particular, a specialization of the approach can be used in application domains where queries are represented as finite state automata (FSAs) or regular expressions (REs) and the database is a collection of FSAs or REs, with non-empty FSA intersection or regular language intersection as the query operator. Example domains include user profile matchmaking and XML filter maintenance
[Chan et al., 2002, Diao et al., 2002, Chan et al., 2003]. With user profile matchmaking, we assume that user profiles are expressed as REs that are constructed based on a global XML schema, meaning we assume XML-based database sources [1]. This assumption is reasonable because today almost all current database systems offer tools to export or view their content in XML format [Lehti and Fankhauser, 2004]. REs express parts of an XML document graph certain users are interested in for example. By comparing REs belonging to two users, we can check if the two

[1] User profiles are modeled as Xpath expressions in [Altinel and Franklin, 2000], and Xpath expressions are a special class of REs.

users have common interests by checking for non-emptiness of RE languages. For large user profile repositories, it becomes interesting how to index such RE data collections to support RE matchmaking. With filter maintenance, filters to incoming XML documents are expressed using REs. The filter maintenance objective is to be able to check if a given input filter is already in a collection, for example before adding it to the collection. This operation is performed by comparing the languages of REs representing the input filter and REs for filters already in the collection for non-emptiness. If the filter collection is very large, the filters must be indexed for efficient querying. However, standard indexes cannot represent REs for evaluation of intersection and non-emptiness query operations.

Another potential application domain for the approach described in this dissertation is the unification of business processes in case of company mergers. As an example, if two companies are to be merged, it is sometimes necessary to streamline the different departments, some of which might be performing the same functions like sales. In large multi-national companies, it might turn out that several sales departments with different processes might exist, for example specializing on different products or based on different geographical locations. The problem of matching and streamlining such processes for more than one large multi-national company is non-trivial. The approach described in this dissertation can be used to find matching business processes from the same role by relaxing the complementarity requirement for roles. The matching condition $role_1 \cap role_2 = \emptyset$ is relaxed since the complementarity of roles is no longer required.

6.2 Future Work

Future work can be construed along the following lines: (i) semantic matchmaking of complex business processes, (ii) approximate matchmaking of complex business processes (iii) matchmaking of business processes with ranking (iv) application to other workflow formalisms like Petri nets and bisimilarity measures and (v) matchmaking business processes with complex cycles.

6.2.1 Semantic Matchmaking of Complex Business Processes

Semantic matchmaking of complex business processes and their efficient implementations are being addressed by the semantic Web community. There currently exists two major initiatives aimed at describing Web services semantically to facilitate the automated discovery, composition, selection, execution, and monitoring of Web services. Both groups have made submissions to the W3C for standardization. The two major standards are the Web Service Modeling Ontology (WSMO) standard [Roman et al., 2005, Lausen et al., 2005] and standards around it and Web Ontology Language (OWL)-based Web Service Ontology (OWL-S) [Martin et al., 2004, Coalition, 2004]. The two groups are in the process of developing ontologies for semantically

describing different aspects of Web services. Semantic descriptions of various aspects of Web services will no doubt facilitate machine processing of Web services, including service discovery. To this day, both initiatives give ontology descriptions of the various aspects of Web services, including workflow or business process related aspects; for example, choreography descriptions of Web services in WSMO. None of these descriptions provide a mechanism how to efficiently find matching Web services based on ontological descriptions of their business processes within a large service repository. The WSMO standardization process is still on-going and especially the choreography mechanism is not yet fully developed. An interesting research challenge is how to provide an efficient mechanism for service discovery based on workflow or business process-related ontological descriptions of Web services within WSMO? While OWL-S has been around for a longer period (initially called DAML-S) than WSMO, it faces the same challenges. The mechanism for efficient service discovery based on dimensions like ontological descriptions of process aspects of Web services are not available within OWL-S.

Semantic aspects of service discovery must be fully defined as well as semantic descriptions of Web service functions. The problem of semantic description of a Web service function is the need for universal agreement on the semantics. This is a challenge that is yet to be overcome. Future work must address these challenges.

6.2.2 Approximate Matchmaking of Complex Business Processes

Approximate matchmaking of complex business processes requires the development of similarity measures for estimating the degree to which given business process descriptions related to each other. These similarity measures can be used to find which business process descriptions fall within a given threshold, thus act as benchmarks by which to compare business process descriptions. The similarity measures must be based on a formal model such as annotated finite state automata or Petri Nets. The similarity measures can be based on the structure of the formal model description of a business process for example states, transitions, final states etc., within the formal model or issues like business process semantics based on some ontological descriptions. Approximate matchmaking of business processes is important because extreme measures can be taken - which business process descriptions are equivalent and which are completely unrelated. Approximate measures are useful in a business sense, because sometimes it is important to find which business process descriptions match within a given threshold, after which manual corrective action can be taken if necessary to align the different descriptions. Future work can focus on developing these similarity measures and efficient mechanisms for searching Internet scale repositories based on the similarity measures.

6.2.3 Matchmaking of Business Processes with Ranking

Further future work related to the work presented in this dissertation is the development of ranking mechanisms for search results. If multiple results are returned during a search operation, the

service finder would be interested in finding out which results match more closely to his or her query. This calls for the development of a measure to determine the closeness of matchmaking results, thus this research issue is closely related to that described in Section 6.2.2.

6.2.4 Application to Other Workflow Formalisms

Another aspect of future work is how to apply the presented indexing approach to other workflow formalisms such as Petri nets and bisimulation. As an example, in this dissertation, we have extended finite state automata so that they can handle mandatory messages of a business process. The question is, to what extend can this be applied to other formalisms such as Petri Nets and bisimulation? To what extend can other measures, other than intersection be used for business process matchmaking? Future work can explore how to vary some aspects of our approach and rely on other formalisms to formally specify business process descriptions and define efficient implementations based on the formal descriptions.

6.2.5 Matchmaking Business Processes with Complex Cycles

In this dissertation we have shown that annotated finite state automata with complex cycles needed a different approach for matchmaking. We have noted that this is not a critical issue for business processes, because in practice business process descriptions comprise simple cycles. However, the handling of complex cycles remains an open question. Our experience in this research showed that complex cycles in aFSAs introduce high rates of false matches; at the same time, it is not possible to increase the look-back significantly due to the high complexity of constructing the message sequences, thus making it difficult to get rid of the false matches. Thus other approaches must be considered for those applications where complex cycles are an inherent part of the process specification. Future work will look into this problem.

6.3 Final Remarks

The research work presented in this dissertation contributes to the state of the art of service discovery in Web service infrastructures in an important way. This research will make it possible for the first time, to use business process descriptions as another dimension for service discovery within the Web service infrastructure. We have shown in the dissertation how our approach can be used to extend the current UDDI service discovery infrastructure by allowing querying based on business process descriptions, and how the results of UDDI searching are merged with those for business process matchmaking using natural join operations. The dissertation also presented an indexing mechanism for efficient implementation of our formal semantics to support large service repositories like those likely to be found on the Internet. Detailed evaluations and

analysis have shown that our indexing approach scales well with increasing data set sizes and outperforms sequential scanning by several orders of magnitude. In the dissertation, it was also shown that a specialization of our indexing approach can be used in other domains such as user profile matching, which is useful for resource distribution, targeted on-line marketing, recommender systems etc. Evaluation and analysis of the specialization of our index shows that our indexing approach has better best and worst case performances than RE-tree index [2] for specifications whose automata have simple cycles. Thus a specialization of our index provides a practical alternative to the RE-tree index since most of the addressed problems can be modeled by automata with simple cycles.

[2] RE-tree index is a special index for indexing and searching regular expressions

Bibliography

[Aissi et al., 2002] Aissi, S., Chan, A., Clark, J. B., Fischer, D., Fletcher, T., Hayes, B., Kartha, N., Liu, K., Malu, P., Moberg, D., Mukkamala, H., Ogden, P., Sachs, M., Saito, Y., Smiley, D., Weida, T., Wenzel, P., and Zheng, J. (2002). Collaboration-protocol profile and agreement specification version 2.0. http://www.ebxml.org/specs/ebcpp-2.0.pdf.

[Altinel and Franklin, 2000] Altinel, M. and Franklin, M. J. (2000). Efficient filtering of XML documents for selective dissemination of information. In *The VLDB Journal*, pages 53–64.

[Andrews et al., 2003] Andrews, T., Curbera, F., Dholakia, H., Goland, Y., Klein, J., Leymann, F., Liu, K., Roller, D., Smith, D., Thatte, S., Trickovic, I., and Weerawarana, S. (2003). Business process execution language for web services, version 1.1. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

[Ariba et al., 2000] Ariba, IBM, and Microsoft (2000). Universal description, discovery and integration. http://www.uddi.org/.

[Baeza-Yates, 1992] Baeza-Yates, R. A. (1992). Text retrieval: theory and practice. In van Leeuwen, J., editor, *Proceedings of the 12th IFIP World Computer Congress*, pages 465–476, Madrid, Spain. North-Holland.

[Banerjee et al., 1987] Banerjee, J., Chou, H., Garza, J. F., Kim, W., Woelk, D., Ballou, N., and Kim, H. (1987). Data model issues for object-oriented applications. *ACM Transactions on Information Systems (TOIS)*, 5(1):3–26.

[Berbers-Lee et al., 2001] Berbers-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific America*, 284(5):34–43.

[Bertino, 1994] Bertino, E. (1994). Index configuration in object-oriented databases. *The VLDB Journal, The International Journal on Very Large Data Bases*, 3(3):355–399.

[Bertino et al., 1998] Bertino, E., Catania, B., and Chiesa, L. (1998). Definition and analysis of index organizations for object-oriented database systems. *Information Systems*, 23(2):65–108.

[Bertino and Foscoli, 1995] Bertino, E. and Foscoli, P. (1995). Index organizations for object-oriented database systems. *TKDE*, 7(2):193–209.

[Bertino and Kim, 1989] Bertino, E. and Kim, W. (1989). Indexing techniques for queries on nested objects. *IEEE Transactions on knowledge and data engineering*, 1(2).

[Bruijn, 2005] Bruijn, J. D. (2005). The web service modeling language wsml. http://www.wsmo.org/TR/d16/d16.1/v0.2/20050320/.

[Buneman, 1997] Buneman, P. (1997). Semistructured data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 117–121. ACM Press.

[Buneman et al., 1997] Buneman, P., Davidson, S., Fernandez, M., and Suciu, D. (1997). Adding structure to unstructured data. In Afrati, F. N. and Kolaitis, P., editors, *Database Theory—ICDT'97, 6th International Conference*, volume 1186, pages 336–350, Delphi, Greece. Springer.

[Bunke, 2000] Bunke, H. (2000). Recent developments in graph matching. In *International conference on pattern recognition (ICPR'00)*.

[Burdett, 2000] Burdett, D. (2000). Internet open trading protocol - iotp - version 1.0. http://www.ietf.org/rfc/rfc2801.txt.

[Chan et al., 2003] Chan, C., Garofalakis, M., and Rastogi, R. (2003). Re-tree: an efficient index structure for regular expressions. *The VLDB Journal*, 12(2):102–119.

[Chan et al., 2002] Chan, C. Y., Felber, P., Garofalakis, M. N., and Rastogi, R. (2002). Efficient filtering of XML documents with XPath expressions. *The VLDB Journal*, 11:354–379.

[Chappell et al., 2001] Chappell, D. A., Chopra, V., Dubray, J., van der Eijk, P., Evans, C., Harvey, B., McGrath, T., Nickull, D., Noordzij, M., Peat, B., and Vegt, J. (2001). *Professional ebXML Foundations*. Wrox Press Inc.

[Chawathe et al., 1994] Chawathe, S. S., Chen, M., and Yu, P. S. (1994). On index selection schemes for nested object hierarchies. In Bocca, J. B., Jarke, M., and Zaniolo, C., editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 331–341. Morgan Kaufmann.

[Chiat et al., 2004] Chiat, L. C., Huang, L., and Xie, J. (2004). Matchmaking for semantic web services. In *Services Computing, 2004 IEEE International Conference on (SCC'04)*, pages 455 – 458.

[Chomicki and Saake, 1998] Chomicki, J. and Saake, G., editors (1998). *Logics for Database and Information Systems*. Kluwer.

[Coalition, 2004] Coalition, T. O. S. (2004). Owl-s 1.1 beta release. http://www.daml.org/services/owl-s/1.1B.

[Comer, 1979] Comer, D. (1979). The ubiquitous b-tree. *Computing surveys*, 11(2).

[Cooper et al., 2001] Cooper, B., Sample, N., F.lin, M. J., Hjaltason, G. R., and Shadmon, M. (2001). A fast index for semistructured data. In *The VLDB Conference*, pages 341–350.

[Cooper and Shadmon, 2000] Cooper, B. and Shadmon, M. (2000). The index fabric: Technical overview. Technical report, RightOder Inc. http://www.rightorder.com/technology/overview.pdf.

[Cordella et al., 1998] Cordella, L., Foggia, P., Sansone, C., Tortorella, F., and Vento, M. (1998). Graph matching: a fast algorithm and its evaluation.

[Dean and Schreiber, 2004] Dean, M. and Schreiber, G. (2004). Owl web ontology language reference, w3c recommendation. http://www.w3.org/TR/owl-ref.

[Diao et al., 2002] Diao, Y., Fischer, P., F.lin, M. J., and To, R. (2002). Yfilter: Efficient and scalable filtering of xml documents. In *Proceedings of the 18th international conference of data engineering (ICDE 2002)*, pages 341–342. IEEE Computer Society.

[Dong et al., 2004] Dong, X., Halevy, A., Madhavan, J., Nemes, E., and Zhang, J. (2004). Similarity search for web services. *In Proc. of VLDB, 2004*.

[EBXML, 2005] EBXML (2005). ebXML standardization. http://www.ebxml.org/.

[Esparza, 1998a] Esparza, J. (1998a). Decidability and complexity of petri net problems - an introduction. *In G. Rozenberg and W. Reisig, editors, Lectures on Petri Nets I: Basic Models. Advances in Petri Nets, number 1491 in Lecture Notes in Computer Science*, pages 374–428.

[Esparza, 1998b] Esparza, J. (1998b). Reachability in live and safe free-choice petri nets is np-complete. *Theoretical Computer Science*, 198(1-2):211–224.

[Ginsburg, 1975] Ginsburg, S. (1975). *Algebraic and automata-theoretic properties of formal languages*. North-Holland/ American elsevier.

[Giugno and Shasha, 2002] Giugno, R. and Shasha, D. (2002). Graphgrep: A fast and universal method for querying graphs. In *16th International Conference in Pattern recognition (ICPR)*, Quebec, Canada. IEEE Computer Society.

[Goldman et al., 1999] Goldman, R., McHugh, J., and Widom, J. (1999). From semistructured data to XML: Migrating the lore data model and query language. In *Workshop on the Web and Databases (WebDB '99)*, pages 25–30.

[Goldman and Widom, 1997] Goldman, R. and Widom, J. (1997). Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23rd VLDB Conference*, Athens, Greece.

[Gonzalez-Castillo et al., 2001] Gonzalez-Castillo, J., Trastour, D., and Bartolini, C. (2001). Description logic for matchmaking of services. Technical Report HPL-2001-265, Hewlett-Packard.

[Gray and Reuter, 1993] Gray, J. and Reuter, A. (1993). *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, California.

[Gudes, 1997] Gudes, E. (1997). A uniform indexing scheme for object-oriented databases. *Information Systems*, 22(4):199–221.

[Gudgin et al., 2003] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J., and Nielsen, H. F. (2003). Soap version 1.2 part 1: Messaging framework.

[Guttman, 1984] Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, Boston, Mass.*

[Gyssens et al., 1990] Gyssens, M., Paredaens, J., and van Gucht, D. (1990). A graph-oriented object database model. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 417–424. ACM Press.

[Haarslev et al., 2005] Haarslev, V., Mller, R., and Wessel, M. (2005). Racer. http://www.sts.tu-harburg.de/ r.f.moeller/racer/.

[Hellerstein and Pfeffer, 1994] Hellerstein, J. M. and Pfeffer, A. (1994). The rd-tree: An index structure for sets. Technical report, University of Wisconsin, Computer Science Department. Technical Report, 1252.

[Helmer, 1997] Helmer, S. (1997). Index structures for databases containing data items with setvalued attributes. Technical Report Report 2/97, Universitat Mannheim. http://pi3.informatik.uni-mannheim.de.

[Helmer and Moerkotte, 2003] Helmer, S. and Moerkotte, G. (2003). A performance study of four index structures for set-valued attributes of low cardinality. *The VLDB Journal*, 12(3):244–261.

[Henzinger et al., 1995] Henzinger, M. R., Henzinger, T. A., and Kopke, P. W. (1995). Computing simulations on finite and infinite graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 453–462.

[Hoare, 1985] Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice Hall International.

[Hofreiter et al., 2002] Hofreiter, B., Huemer, C., and Klas, W. (2002). ebxml: Status, research issues, and obstacles. In *Proc. 12th International Workshop on Research Issues in Data Engineering: Engineering E-Commerce/E-Business Systems (RIDE'02)*.

[Homepage, 2005a] Homepage, B. P. (2005a). Binding point. http://www.bindingpoint.com, last visited: 23 February 2005.

[Homepage, 2005b] Homepage, G. C. (2005b). Grand central homepage. http://www.grandcentral.com, last visited: 23 February 2005.

[Homepage, 2005c] Homepage, S. C. (2005c). Sal central homepage. http://www.salcentral.com, last visited: 23 February 2005.

[Homepage, 2005d] Homepage, W. (2005d). Woogle –– web service search engine. http://data.cs.washington.edu/webService, last visited: 23 February 2005.

[Homepage, 2005e] Homepage, W. S. L. (2005e). Web service list homepage. http://www.webservicelist.com, last visited: 23 February 2005.

[Hopcroft et al., 2001] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley.

[Horrocks et al., 2003] Horrocks, I., McGuinness, D. L., and Welty, C. A. (2003). Digital libraries and web-based information systems. pages 427–449.

[Horrocks et al., 2001] Horrocks, I., van Harmelen, F., and Patel-Schneider, P. (2001). Daml+oil. http://www.daml.org/2001/03/daml+oil-index.html.

[IBM et al., 2002] IBM, Microsoft, HP, Oracle, Intel, and SAP (2002). Universal description, discovery and integration. http://www.uddi.org/.

[Jiang et al., 1994] Jiang, Y., Liu, X., and Bhargava, B. (1994). Re-evaluating indexing schemes for nested objects. In *Proceedings of the third international conference on Information and knowledge management*, pages 439–446. ACM Press.

[JUDDI, 2003] JUDDI (2003). juddi. http://ws.apache.org/juddi.

[Kang et al., 2000] Kang, U. T., Davis, K. C., and Ravishankar, S. (2000). Indexing inheritance and aggregation. In *CIKM 2000*.

[Kaushik et al., 2002] Kaushik, R., Bohannon, P., Naughton, J. F., and Korth, H. F. (2002). Covering indexes for branching path queries. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 133–144. ACM Press.

[Kim and Shawe-Taylor, 1994] Kim, J. Y. and Shawe-Taylor, J. (1994). Fast string matching using an *n*-gram algorithm. *Software – Practice and Experience*, 24(1):79–88.

[Kim et al., 1989] Kim, W., Kim, K., and Dale, A. (1989). Indexing techniques for object-oriented databases. *Object-oriented concepts, databases, and applications*, pages 371–394.

[Knuth, 1998] Knuth, D. (1998). *The art of computer programming, Vol. III, sorting and searching, third edition*. Addson Wesley, Reading, MA.

[Kotok and Webber, 2001] Kotok, A. and Webber, D. (2001). *ebXML: The new global standard for doing business on the Internet*. New Riders Publishing.

[Lausen et al., 2005] Lausen, H., Polleres, A., and Roman, D. (2005). Web service modeling ontology (wsmo). W3C Member Submission.

[Lehti and Fankhauser, 2004] Lehti, P. and Fankhauser, P. (2004). Xml data integration with owl: Experiences and challenges. In *2004 Symposium on Applications and the Internet (SAINT'04)*. IEEE Computer Society.

[Levine et al., 2001] Levine, P., Clark, J., Casanave, C., Kanaskie, K., Harvey, B., Clark, J., Smith, N., Yunker, J., and Riemer, K. (2001). Business process specification schema. www.ebxml.org/specs/ebBPSS.pdf.

[Lewis and Catlett, 1994] Lewis, D. and Catlett, J. (1994). Heterogeneous uncertainty sampling for supervised learning. In Cohen, W. W. and Hirsh, H., editors, *Proceedings of ICML-94, 11th International Conference on Machine Learning*, pages 148–156, New Brunswick, US. Morgan Kaufmann Publishers, San Francisco, US.

[Li and Horrocks, 2003] Li, L. and Horrocks, I. (2003). A software framework for matchmaking based on semantic web technology. In *WWW 2003*, pages 331 – 339.

[Mahleko et al., 2005a] Mahleko, B., Wombacher, A., and Fankhauser, P. (2005a). A grammar-based index for matching business processes. In *The IEEE International Conference on Web Services (ICWS 2005)*, pages 21–30, Los Alamitos, California. IEEE Computer Society.

[Mahleko et al., 2005b] Mahleko, B., Wombacher, A., and Fankhauser, P. (2005b). Process-annotated service discovery facilitated by an n-gram-bsed index. In W. Cheung, J. H., editor, *The 2005 IEEE International conference on e-Technology, e-Commerce and e-Service*, pages 2–8, Los Alamitos, California. IEEE Computer Society.

[Maier and Stein, 1986] Maier, D. and Stein, J. (1986). Indexing in an object-oriented dbms. In Dittrich, K. R. and Dayal, U., editors, *1986 International Workshop on Object-Oriented Database Systems, September 23-26, 1986, Asilomar Conference Center, Pacific Grove, California, USA, Proceedings*, pages 171–182. IEEE Computer Society.

[Martin et al., 2004] Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., and Sycara, K. (2004). Owl-s: Semantic markup for web services. W3C Member Submission.

[McHugh et al., 1997] McHugh, J., Abiteboul, S., Goldman, R., Quass, D., and Widom, J. (1997). Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66.

[McIlraith et al., 2001] McIlraith, S., Son, T., and Zeng, H. (2001). Semantic web services. *IEEE Intelligent Systems (Special Issue on the Semantic Web)*.

[Mendelson, 1997] Mendelson, E. (1997). *Introduction to Mathematical Logic*. Chapman and Hall.

[Milner, 1982] Milner, R. (1982). *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc.

[Milner, 1999] Milner, R. (1999). *Communicating and mobile systems: the π-calculus*. Cambridge University Press.

[Milo and Suciu, 1999] Milo, T. and Suciu, D. (1999). Index structures for path expressions. *Lecture Notes in Computer Science*, 1540:277–295.

[Morrison, 1968] Morrison, D. R. (1968). Patricia: Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534.

[NAICS, 2005] NAICS, E. C. P. C. (2005). The north american industry classification system (naics). http://www.census.gov/epcd/www/pdf/naicsbch.pdf, last visited: 24 February 2005.

[Nestorov et al., 1997] Nestorov, S., Ullman, J., Wiener, J., and Chawathe, S. (1997). Representative objects: Concise representations of semistructured, hierarchial data. In *ICDE*, pages 79–90.

[Nodine et al., 1999] Nodine, M., Bohrer, W., and Ngu, A. (1999). Semantic multibrokering over dynamic heterogeneous data sources in infosleuth. In *Proc. of the International Conference on Data Engineering*.

[Nodine et al., 2000] Nodine, M. H., Fowler, J., Ksiezyk, T., Perry, B., Taylor, M., and Unruh, A. (2000). Active information gathering in infosleuth. *International Journal of Cooperative Information Systems*, 9(1-2):3–28.

[Ooi et al., 1996] Ooi, B. C., Han, J., Lu, H., and Tan, K. L. (1996). Index nesting : an efficient approach to indexing in object-oriented databases. *The VLDB Journal : The International Journal on Very Large Data Bases*, 5(3):215–228.

[OTA, 2003] OTA (2003). Open travel alliance (ota).

[Overhage and T., 2002] Overhage, S. and T., P. (2002). Ws-specification: Specifying web services using uddi improvements. In Chaudhri, A., Jeckle, M., Rahm, E., and Unland, R., editors, *Web, Web-Services, and Database Systems: NODe 2002, Web- and Database-Related Workshops, Erfurt, Germany*, volume Volume 2593 / 2003, pages 100 – 119. Springer-Verlag Heidelberg. Lecture Notes in Computer Science.

[Paige and Tarjan, 1987] Paige, R. and Tarjan, R. E. (1987). Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989.

[Panti, 1998] Panti, G. (1998). Multi-valued logics. In Gabbay, D. and Smets, P., editors, *Handbook of Defeasible Reasoning and Uncertainty Management Systems*, volume 1: Quantified Representation of Uncertainty and Imprecision, chapter 2, pages 25–74. Kluwer, Dordrecht.

[Paolucci et al., 2002] Paolucci, M., Kawmura, T., Payne, T., and Sycara, K. (2002). Semantic matching of web services capabilities. In *Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 333–347. Springer-Verlag.

[Patil et al., 2004] Patil, A. A., Oundhakar, S. A., Sheth, A. P., and Verma, K. (2004). Meteor-s web service annotation framework. In *WWW '04: Proceedings of the 13th international conference on World Wide Web*, pages 553–562. ACM Press.

[Peterson, 1981] Peterson, J. L. (1981). *Petri Net Theory and the Modeling of Systems*. Prentice-Hall.

[Ramaswamy and Kanellakis, 1995] Ramaswamy, S. and Kanellakis, P. C. (1995). OODB indexing by class-division. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 139–150.

[Roman et al., 2005] Roman, D., Keller, U., Lausen, H., de Bruijn, J., Lara, R., Stollberg, M., Polleres, A., Feier, C., Bussler, C., and Fensel, D. (2005). Web service modeling ontology. *Applied Ontology*, 1(1):77–106.

[RosettaNet, 2005] RosettaNet (2005). Rosettanet homepage. http://www.rosettanet.org. last visited, 15 February 2005.

[Sedgewick, 1998] Sedgewick, R. (1998). *Algorithms in C++*. Addison Wesley.

[ShaiklAli et al., 2003] ShaiklAli, A., Rana, O. F., Al-Ali, R., and Walker, D. W. (2003). Uddie: An extended registry for web services. In *Proceedings of the 2003 Symposium on Applications and the Internet Workshops (SAINT-w03)*. IEEE Computer Society.

[Shasha et al., 2002] Shasha, D., Wang, J. T.-L., and Giugno, R. (2002). Algorithmics and applications of tree and graph searching. In Popa, L., editor, *Symposium on Principles of Database Systems*, pages 39–52. ACM.

[Sivashanmugam et al., 2003] Sivashanmugam, K., Verma, K., Sheth, A., and Miller, J. (2003). Adding semantics to web services standards. In *Proceedings of the 1st International Conference on Web Services (ICWS'03)*, pages 395 – 401.

[SWIFT, 2005] SWIFT (2005). Swift homepage. http://www.swift.com. last visited, 22 March 2005.

[Sycara et al., 1999a] Sycara, K., Lu, J., Klusch, M., and Widoff, S. (1999a). Matchmaking among heterogeneous agents on the internet. In *Proc. of the AAAI Spring Symposium on Intelligent Agents in Cyberspace*.

[Sycara et al., 1999b] Sycara, K. P., Klusch, M., Widoff, S., and Lu, J. (1999b). Dynamic service matchmaking among agents in open information environments. *SIGMOD Record*, 28(1):47–53.

[Szpankowski, 1990] Szpankowski, W. (1990). Patricia tries again revisited. *Journal of the ACM (JACM)*, 37(4):691–711.

[Team, 2001a] Team, B. P. (2001a). Business process specification schema v1.01.

[Team, 2001b] Team, T. P. (2001b). Collaboration-protocol profile and agreement specification v1.0. http://www.ebxml.org/specs/index.htm.

[Trastour et al., 2001] Trastour, D., Bartolini, C., and Gonzalez-Castillo, J. (2001). A semantic web approach to service description for matchmaking of services. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*.

[Ullman, 1988] Ullman, J. D. (1988). *Principles of Database and knowledge-base systems volume I*. Computer Science Press, Rockville, Maryland.

[UNSPSC, 2005] UNSPSC, U. N. O. (2005). The universal standard products and services classification (unspsc). http://www.eccma.org/unspsc_dl.html, last visited: 24 February 2005.

[van der Aalst, 1999] van der Aalst, W. (1999). Interorganizational workflows: An approach based on message sequence charts and petri nets. *Systems Analysis - Modelling - Simulation*, 34(2):335–367.

[van der Aalst and Hee, 2002] van der Aalst, W. and Hee, K. V. (2002). *Workflow Management - Models, Methods, and Systems*. MIT Press.

[van der Aalst, 2003] van der Aalst, W. M. P. (2003). Dont go with the flow: Web services composition standards exposed. *IEEE Intelligent Systems*, pages 72–76.

[Vitter, 2001] Vitter, J. S. (2001). External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys (CSUR)*, 33(2):209–271.

[Weinblatt, 1972] Weinblatt, H. (1972). A new search algorithm for finding the simple cycles of a finite directed graph. *Journal of the Association for Computing Machinery*, 19(1).

[Wombacher, 2005] Wombacher, A. (2005). *Decentralized establishment of consistent, multi-lateral collaborations*. PhD thesis, Technischen Universitt Darmstadt, Darmstadt, Germany.

[Wombacher et al., 2004a] Wombacher, A., Fankhauser, P., Mahleko, B., and Neuhold, E. (2004a). Matchmaking for business processes based on choreographies. *International Journal of Web Services*, 1(4):14–32.

[Wombacher et al., 2004b] Wombacher, A., Fankhauser, P., and Neuhold, E. (2004b). Transforming bpel into annotated deterministic finite state automata for service discovery. In *IEEE International Conference on Web Services (ICWS 2004)*, Los Alamitos, California. IEEE Computer society.

[Wombacher et al., 2004c] Wombacher, A., Mahleko, B., and Neuhold, E. (2004c). IPSI-PF: A business process matchmaking engine. In *Proc. of IEEE Conf. on Electronic Commerce (CEC)*, pages 137–145.

[Wombacher et al., 2005] Wombacher, A., Mahleko, B., and Neuhold, E. J. (2005). IPSI-PF: A business process matchmaking engine based on annotated finite state automata. *Information Systems and E-Business Management*, 3(2):127 – 150.

[WSMX, 2005] WSMX (2005). Web service modelling execution environment home page. http://www.wsmx.org/. Last visited, 28 November 2005.

[Wu et al., 2004] Wu, K., Otoo, E., and Shoshani, A. (2004). On the performance of bitmap indices for high cardinality attributes. In *Proceedings of the 30th VLDB Conference, Toronto, Canada, 2004*.

A Transitions on Final States

The appendix describes an algorithm for removing outgoing transitions from final states of an aFSA. The goal is to ensure that during the traversal of a query aFSA graph (described in Section 4.4.3), all transitions can be reached. If final states have outgoing transitions, some transitions will not be reached as the recursion stops as soon as a final state is found, meaning all transitions coming after the final state will not be visited (see Algorithm A).

The algorithm described in this section creates additional states and ε transitions to remove outgoing transitions from aFSA final states. The transformation of the aFSA does not change the language of the original aFSA, since no transitions are removed, but additional transitions (ε transitions) are added.

Initial Values: $Q' = Q, \Delta' = \Delta, F' = F, Vis = \emptyset$, and $q = q_0$

Algorithm : RFS

```
1:    RFS(q,Vis){
2:          T := {t | source(t) = q};
3:          if (T! = null)
4:               for (t ∈ T)
5:                    if (t ∉ Vis) {
6:                         Vis = Vis ∪ {t};
7:                         if (q ∈ F ∧ transitions(q) ∉ ∅) {
8:                              q' := new State();
9:                              Q' := Q' ∪ {q'};
10:                             F' := F' ∪ {q'};
11:                             F' := F' \ {q};
12:                             t' := new Transition(q, ε, q') ;
13:                             Δ' := Δ' ∪ {t'} ;
14:                             RFS(target(t), Vis) ;
                           }
                    }
   }
```

Let $A = (Q, \Sigma, \Delta, q_0, F, QA)$ be an aFSA; a new aFSA $A' = (Q', \Sigma \cup \varepsilon, \Delta', q_0, F', QA)$ is constructed using algorithm *RFS* above where all out-going transitions on final states are removed. The new tuple elements of A' which are Q', Δ' and F' are obtained from *RFS* as described below.

The set of messages of A' comprises the original message set Σ plus ε, where ε represents an empty string. No new annotations are added to new states, so the set QA is not affected by the transformation. The start state does not change also. Initial values of variables are as follows: the set of states of A' is set to the set of states in A; the set of transitions in A' is also set to the set of transitions in A and the

set of final states is initialized as well to the set of final states in A'. q and Vis are used as input to the function, where q is initialized to q_0 and $Vis = \emptyset$. In Line 2, the outgoing transitions from the current state are computed and added to a collection T. If the collection T is not null, the algorithm iterates over the transitions of T as shown in lines 4 - 13. In line 5, the algorithm checks if the current transition is already visited. If the transition is already visited, the next transition is checked. The termination condition for the recursion is the visited transitions. If a transition is found which is not visited, in line 6 this transition is added to the set of visited transitions. Next in line 7 the current state is checked for membership to F the set of final states as well as whether it has outgoing transitions. If the current state is a final state and has at least one outgoing transition, the following steps are taken: (i) a new state q' is created and added to the set of states Q' as well as to the set of final states F'; (ii) the current state q is removed from the set of final states (iii) a new transition t' is created having q as source state, ε, as the message or transition label and q' as target state and (iv) the new transition t' is added to the set of transitions Δ'. Line 14 is a recursive call to the function RFS, with the target of the current transition t and the new set of visited transitions.

The algorithm described above is a depth first search (DFS) algorithm which visits all transitions only once, performing basic operations to create new states, adding states and transitions to existing collections etc. Being a DFS algorithm it has linear time performance on the number of transitions as computational complexity.

B Publication List

1. Indexing Business Processes based on annotated Finite State Automata. The 2006 IEEE International Conference on Web Services (ICWS 2006), Chicago, USA. B. Mahleko and A. Wombacher, 2006. To appear.
2. A Grammar-Based Index for Matching Complex Services. International Journal of Web Services Research, 2006, B. Mahleko and A. Wombacher and P. Fankhauser (extended version of IEEE ICWS 05 paper), Submitted.
3. A grammar-based index for matching business processes. IEEE ICWS 05. Orlando, Florida, USA. B. Mahleko and A. Wombacher and P. Fankhauser, 2005.
4. Process-annotated Service Discovery facilitated by an n-gram-based index. IEEE EEE 05, Hong Kong. B. Mahleko and A. Wombacher and P. Fankhauser. 2005.
5. IPSI-PF: A Business Process Matchmaking Engine based on annotated finite state automata. A. Wombacher, B. Mahleko and E. Neuhold. Information Systems and E-Business Management, Vol. 3, No. 2, Jul 2005, (extended version of CEC'04 paper).
6. Matchmaking for Business Processes Based on Conjunctive Finite State Automata. International Journal of Business Process Integration and Management. A. Wombacher and P. Fankhauser and B. Mahleko and E Neuhold. Vol. 1. No. 1, 2005. (extended version of CEC'03 paper).
7. eureauweb: An Information System for Users of European Inland Waterways. MDM 2005, Ayia Napa, Cyprus. H. Kirchner and B. Mahleko, S & E Eklington and M. Kelly. 2005.
8. Matchmaking for Business Processes Based on Choreographies. International Journal of Web Services, Vol. 1, No. 4. A. Wombacher, P. Fankhauser, B. Mahleko and E. Neuhold. 2004. (extended version of EEE-04 paper).
9. IPSI-PF: A Business Process Matchmaking Engine. CEC'04, San Diego, California, USA. A. Wombacher, B. Mahleko and E. Neuhold, 2004.
10. Matchmaking for Business Processes based on Choreographies. EEE-04. Taipei, Taiwan, A. Wombacher, P. Fankhauser, B. Mahleko and E. Neuhold, 2004.
11. eureauweb : An Architecture for a European Waterways Networked Information System. ENTER'04, Cairo, Egypt. H. Kirchner, B. Mahleko, M Kelly, R. Krummenacher and Z. Wang, 2004.
12. Classification of ad-hoc multi-lateral collaborations based on local workflow models. ACM SAC 2003, Melbourne, Florida, USA. 1185 : 1190, A. Wombacher, B. Mahleko and T. Risse, 2003.
13. Efficient Match-Making of Business Processes. CAiSE 2003, 10th Doctoral Consortium, Velden, Austria. B. Mahleko, 2003.

14. Matchmaking for Business Processes. CEC 2003, Newport Beach, California, USA. 7-11. A.Wombacher, P. Fankhauser, B. Mahleko and E. Neuhold, 2003.
15. Ad-Hoc Business Processes in Web Services. SAINT Workshops 2003, Orlando, Florida, USA. 101-105, A. Wombacher and B. Mahleko, 2003.
16. Finding Trading Partners to Establish Ad-hoc Business Processes. CoopIS/ DOA/ ODBASE 2002, Irvine, California, USA. 339-355. A. Wombacher and B. Mahleko, 2002.
17. Towards a Platform for Supporting the Buyer in Trading in Heterogeneous Marketplaces. WECWIS 2002, Newport Beach, California, USA. 249-251, B. Mahleko, J. Klingemann, P. Fankhauser and A. Wombacher, 2002.
18. Intelligent Web Service - From Web Services to .Plug & Play. Service Integration OTM Conferences (1) 2005, Agia Napa, Cyprus. 18-19. E. Neuhold, T. Risse and A. Wombacher, C Niedere and B Mahleko, 2005.
19. From Call for Tenders to Sealed-Bid Auction for Mediated Ecommerce. DS-9 2001, Hong Kong. 69-86, O. Tafreschi, M. Schneider, P. Fankhauser, B. Mahleko and T. Tesch, 2001.

C Trademarks

RosettaNet PIP is a trademark or registered trademark of RosettaNet.
Open Travel is a trademark or registered trademark of the Open Travel Alliance.
DUNS is a registered trademark of The Dun & Bradstreet Corporation.
Java and J2EE are trademarks or or registered trademarks of Sun Microsystems, Inc.
ebXML is a trademark or registered trademark of OASIS.
UNSPSC is a trademark of the United Nations/UNDP
JBoss is a trademark of Marc Fleury under the JBoss Group.
Pentium Processor is a trademark or registered trademark of Intel Corporation.
Windows is a trademark or registered trademark of Microsoft Corporation.
Mathematica is a registered trademark of Wolfram Research, Inc.

D Curriculum Vitae

PERSONAL INFORMATION

Name	Mahleko, Bendick
Date of Birth	23 February, 1970
Place of Birth	Chipinge, Zimbabwe

EDUCATION

04/1996 - 12/1997	Masters degree (MSc), Computer Science, National University of Science and Technology, Bulawayo, Zimbabwe
03/1992 - 12/1995	Bachelors degree honours (BSc, hons), Computer Science, National University of Science and Technology, Bulawayo, Zimbabwe

WORK EXPERIENCE

02/2000 - 02/2006	Research Associate, Fraunhofer Research Institute, Darmstadt, Germany
01/1998 - 01/2000	Lecturer, Department of Computer Science, National University of Science and Technology, Bulawayo, Zimbabwe
03/1992 - 12/1995	Bachelors degree honours (BSc, hons), Computer Science, National University of Science and Technology, Bulawayo, Zimbabwe